UiO : Department of Informatics
University of Oslo

A Software Defined Networking evaluation approach to distributing load

Using POX, Floodlight and BIG-IP 1600

Emil Sylvio Golinelli

Master's Thesis Spring 2015

A Software Defined Networking evaluation approach to distributing load

Emil Sylvio Golinelli

May 18, 2015

Abstract

SDN is believed to be the next big thing in networking, by reducing cost and replacing different networking units. However, how much of a leap is SDN moving forward? Could it for instance replace a dedicated load-balancing unit? In this thesis a BIG-IP 1600 series load balancer is measured against different SDN-based load balancing techniques. The two most fundamental differences in approaches are *proactive* and *reactive* both of which are tested. Because of a hard software limit, the available controllers POX and Floodlight struggle with the performance of their reactive implementations. Due to the similar hard limit, at this point it is unknown if better developed controllers can solve this issue. It is however most likely. One thing to note is that the performance of the load balancing will be totally dependent on the system resources the controller has available. For the other approach, the results show that the viability of proactive implementations replacing a dedicated unit is more promising if the complexity and flow updating issues can be resolved.

Contents

I Introduction **1**

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Problem Statement . . . 4
- 1.3 Thesis Structure . . . 5
- 1.4 Related Work . . . 6
  - 1.4.1 OpenFlow-Based Server Load Balancing Gone Wild . . . 6
  - 1.4.2 OpenFlow Based Load Balancing . . . 6
  - 1.4.3 Aster*x: Load-Balancing as a Network Primitive . . . 6
  - 1.4.4 Load Balancing in a Campus Network using Software Defined Networking . . . 7

2 Background **9**
- 2.1 The OSI model . . . 9
  - 2.1.1 Layer 1 . . . 9
  - 2.1.2 Layer 2 . . . 9
  - 2.1.3 Layer 3 . . . 10
  - 2.1.4 Layer 4 . . . 10
  - 2.1.5 Layer 7 . . . 10
- 2.2 Traditional Networks . . . 10
  - 2.2.1 Networking devices . . . 11
  - 2.2.2 Hub . . . 11
  - 2.2.3 Switch . . . 11
  - 2.2.4 Router . . . 11
- 2.3 Load balancer . . . 12
- 2.4 Hardware load balancing . . . 12
  - 2.4.1 F5 hardware load balancer . . . 13
- 2.5 SDN load balancing . . . 14
- 2.6 Software Defined Networking (SDN) . . . 16
  - 2.6.1 Application Layer . . . 17
  - 2.6.2 Control Layer . . . 17
  - 2.6.3 Infrastructure Layer . . . 18
- 2.7 OpenFlow . . . 18
  - 2.7.1 Network Flows . . . 19
  - 2.7.2 OpenFlow Flow Table . . . 20
  - 2.7.3 OpenFlow PipeLine . . . 20
  - 2.7.4 OpenFlow versions . . . 22

2.8 OpenFlow switch . . . 22
2.9 OpenFlow controllers . . . 23
2.9.1 POX . . . 23
2.9.2 Floodlight . . . 24
2.10 Testbeds . . . 24
2.10.1 Mininet . . . 25
2.11 Benchmarking/assisting tools . . . 26
2.11.1 Iperf . . . 26
2.11.2 Wireshark . . . 26
2.11.3 Tcpdump . . . 26
2.11.4 Oracle VM VirtualBox . . . 27
2.11.5 ethstats . . . 28
2.11.6 httperf . . . 28
2.11.7 ApacheBench (ab) - Apache HTTP server benchmarking tool . . . 29

II The project **31**

3 Planning the project **33**
3.1 Testbed design . . . 33
3.1.1 Virtual environment with Mininet . . . 33
3.2 Hardware environment build . . . 36
3.2.1 Configuration of servers and clients . . . 36
3.2.2 GUI configuration of BIG-IP 1600 . . . 38
3.3 Load Balancing methodology for SDN . . . 41
3.4 Comparing SDN controller solutions . . . 42
3.4.1 POX SDN controller . . . 42
3.4.2 Floodlight SDN controller . . . 43
3.5 Experiments methodology . . . 44
3.6 Experiments Evaluation . . . 44
3.6.1 Evaluating the performance of the solutions . . . 44
3.6.2 Evaluating the scalability of the solutions . . . 45
3.7 Experiments . . . 46

III Conclusion **47**

4 Results and Analysis **49**
4.1 Results of initial link performance tests . . . 49
4.1.1 Mininet network configuration of links . . . 50
4.1.2 Mininet CPU limits . . . 51
4.1.3 Mininet configuration file for run-time parameters . . . 51
4.2 Performance test results for the physical environment . . . 52
4.2.1 httperf tests . . . 52
4.2.2 ab tests . . . 52
4.3 Performance test results for the virtual environment . . . 55
4.3.1 Mininet performance issue . . . 55

4.3.2 POX Controller 56
4.3.3 Floodlight Controller 60
4.3.4 SDN-based Load Balancing Results 61
4.3.5 Proactive POX load balancing 63
4.4 Solutions comparison 65
4.5 Solutions analysis 67

5 Discussion and Future Work **69**
5.1 Discussion and Conclusion 69
5.1.1 Network topology 69
5.1.2 Feasibility cases for SDN 69
5.1.3 Load Balancing algorithms 70
5.1.4 Mininet bug, implications on recorded data 71
5.1.5 MiB instead of MB 71
5.1.6 Problem statement 72
5.2 Recommendations for Future Work 73

Appendices **79**

A Mininet scripts **81**

B Raw data **89**

List of Figures

2.1 OSI-model (Zito, 2013) 10
2.2 Load Balancer example for web-traffic balancing 13
2.3 SDN Load Balancing in a OpenFlow network using a Reactive approach. 15
2.4 SDN Load Balancing in a OpenFlow network using a Proactive approach. 15
2.5 SDN model (open networking foundation, 2015) 16
2.6 OpenFlow enabled network vs traditional 19
2.7 OpenFlow match fields for packets. 20
2.8 The pipeline of OpenFlow (OpenFlow, 2013). (a) for the whole pipeline. (b) for one table in the pipeline 21
2.9 Floodlight GUI at start page 25
2.10 WireShark GUI with OpenFlow traffic packets captured . . . 27
2.11 VirtualBox GUI for Mininet instalation 27

3.1 Testbed virtual setup 34
3.2 Final hardware setup. 37
3.3 Configuration of networking bridge of the two VLANs in the F5 GUI setup 39
3.4 Configuration of self IP in the F5 GUI setup 39
3.5 Inner workings of the F5 VLAN bridge and the Load Balancing module. 40
3.6 Health monitoring properties for the pool. 40
3.7 Pool members and load balancing algorithm. 41
3.8 Virtual IP settings page. 42

4.1 average latency comparison 50
4.2 BIG-IP calculated saturation, after formula: $434.15x - 3223.52$. Showing RPS and CPU usage. 54
4.3 XTerm CLI starting HTTP server 55
4.4 SDN as a Load Balancing algorithm results 62
4.5 SDN as a Load Balancing algorithm results for large load . . 63
4.6 Proactive load balancing POX for few bytes in httperf. Figure in base10, shows relativity between CPU usage of switch and RPS achieved. 66
4.7 Proactive and reactive performance diagram for httperf tests of few bytes 66

4.8 Proactive and reactive performance diagram for httperf tests of 1MiB 67

List of Tables

2.1 Specifications for the BIG-IP 1600 series 14
2.2 OpenFlow table example 20
2.3 OpenFlow entries columns (OpenFlow, 2013). 20

3.1 Specifications for the virtual environment 34

4.1 BIG-IP, results for few bytes (94) 53
4.2 BIG-IP, results for few bytes (94) CPU load 53
4.3 BIG-IP, results for 1 MiB 55
4.4 POX Controller, results for few bytes (215) AB 57
4.5 POX Controller, results for few bytes (215) httperf 57
4.6 POX Controller, results(few bytes) httperf CPU percentage . 58
4.7 POX Controller, results for 1 MiB httperf 58
4.8 POX Controller, results for 1 MiB CPU usage httperf 59
4.9 POX Controller, results for 2 MiB httperf 59
4.10 POX Controller, results for 2 MiB CPU usage httperf 59
4.11 Floodlight Controller, results for few bytes (215) ab 60
4.12 Floodlight Controller, results for few bytes (215) httperf . . . 60
4.13 Floodlight Controller, results(few bytes) httperf CPU percentage 61
4.14 Floodlight Controller, results for 1MiB httperf 61
4.15 Floodlight Controller, results for 1MiB CPU usage httperf . . 61
4.16 Proactive POX httperf test of 1 server at saturation point - Request rate requested: 42 (req/s), B.1 64
4.17 Proactive POX httperf test of 2 servers at saturation point - Request rate requested: 84 (42x2) (req/s), B.2 65
4.18 Proactive POX httperf test of servers at saturation point for 1 MiB 65

B.1 Proactive POX httperf test of 1 server-raw 90
B.2 Proactive POX httperf test of 2 servers-raw 91

Preface

I would like to thank the following people for their support with my master thesis:

- Desta Haileselassie Hagos for his support and help.
- Kyrre Begnum for providing the hardware load balancers.
- My friends, *Arne* and *Samrin* for helping me proof-read my thesis.

Finally, I would like to extend my gratitude to HiOA and UiO for five rewarding years as a student, and for the opportunity to attend the network and system administration programme.

Part I

Introduction

Chapter 1

Introduction

The introduction chapter begins with the motivation section where it discusses the problems of today's networking, including how SDN can fix those problems and introduce load balancing.

Next section covers the main problem statements this report takes on to solve, before the thesis structure section defines the structure of the next chapters. In the end there is the related work section, which introduces other related work with regards to the problem statement.

1.1 Motivation

After visiting the LISA conference of 2014, where Software Defined Networking (SDN) was one of the hot talks, it became clear that this technology is going to be the next big thing in networking. This is due to the fact that SDN has promises in cost reduction, reduced complexity, better agility, more fine-grained security and simplification for network administrators. One of my criteria for choosing a master thesis was that it had to be useful in regards to working in the sysadmin field after graduation. Getting to work with new and exciting technology as SDN fits that criterion.

What is happening now in the network industry is the same that happened in the early stages of the computer industry. In the beginning, IBM was computing with highly specialized hardware and software from one vendor. That all changed with the introduction of microprocessors, as that introduced many different operating systems (McKeown, 2011) to run on them. These names as known by everyone today are systems like Windows, Linux and Mac OS, and they can run a limitless possibility of applications.

The same thing is now happening with SDN, as it shows promise in including what was earlier a job for dedicated hardware into the network itself. What SDN can achieve may redefine the norm of how networks

are built, as there already is a transition from traditional networking to SDN with the OpenFlow approach in particular. However, there are more and more vendors expanding into SDN and providing their own SDN approaches. In regards to the computer industry analogy, networks may now run their own operation system on common hardware, instead of being limited to a specific provider.

What is good about SDN is that it opens up a possibility for simplification for network administrators. By reducing the complexity for the parities involved, it provides an easier way into the understanding of networking. A way to look at SDN is that it provides an API for interaction with the network. With such an API, applications may be available to the end users in such an easy way as we are getting accommodated to by app stores on our phones (Johnson, 2012). A fully defined and provisioned SDN standard can support apps to be installed with a push of a button. Configuration can therefore be user specific as the SDN framework handles the complexity. For a business setting up a new service, the network may be reconfigured with ease instead of weeks of planning and box-to-box reconfiguring to support the new service.

In summary SDN will shape networking in the following ways (McKeown, 2011):

1. It will empower the people involved with networking, as it can be customized to fit local needs and eliminate unneeded features. It may also include desired features like load balancing.
2. It will increase the speed of innovation, as it provides innovation at the speed of which software is developed as in comparison to hardware development. For instance is rapid development available in emulated environments like Mininet, with the ability to push virtualization developed code directly into the live physical networks with no changes to the code.
3. It will diversify the supply chain, as more vendors and software suppliers all can contribute to the same platform.
4. It will build a strong foundation for abstracting the networking control.

1.2 Problem Statement

Managing traffic in computer networks is very important and critical to ensure a reliable service. This is because traffic often has a high fluctuation, which could result in overloaded devices and an unreliable service for the users. Just like a webserver, a service can only have a limited amount of simultaneously users. If the limit has been reached, no additional users can access the server. Load balancing is a solution that enables more users to access the service, but it is a challenge for network administrators.

Today's solutions works, but it requires expensive dedicated hardware that does not support fast changes in the network topology. BIG-IP is some of the working load balancers products that deliver load balancing via dedicated hardware. These load balancers come at a hefty price tag and must be replaced when maximum capacity is reached. A Software Defined Networking (SDN) load balancer could replace the dedicated hardware, and instead run on commodity hardware.

The question that needs to be answered from a research point of view, is if an SDN solution can match the dedicated hardware.

This thesis tries to address the problems as in the following scenario:

1. How to setup a load balancing algorithm in SDN in order to achieve performance comparable to a physical load balancer.

   (a) Study, setup and run a physical load balancer (BIG-IP-1600 series).

   (b) Compare a SDN-based load balancer to a hardware load balancer in terms of scalability and performance.

1.3 Thesis Structure

- Chapter 1: This is the introduction chapter that introduces the motivation of this thesis and describes the problem statement.

- Chapter 2: The background chapter contains information about SDN, load balancing, tools and terms related to the understanding of these subjects.

- Chapter 3: This is the approach chapter. In this chapter, there is an explanation of the methodology involved when tackling the problem statement. In simpler words: It is a guideline for *how* to attempt to solve the *what*.

- Chapter 4: This chapter contains the actual outcome of the methods applied from the approach, and the analysis of the presented data.

- Chapter 5: This chapter is a discussion of the results in the previous chapter, as it looks back and concludes the thesis. For Future Work it tries to give ways to continue the research in the same field as well as addressing the research questions from the problem statement.

- Appendix: Finally, the appendix chapter includes significant documents related to the thesis.

1.4 Related Work

SDN is a hot topic for research these days; meaning that there are many published papers about different SDN related investigation. Some of those that are related to the problem statement is discussed next.

1.4.1 OpenFlow-Based Server Load Balancing Gone Wild

Richard Wang, Dana Butnariu, and Jennifer from the Rexford Princeton University (Richard Wang, 2011) have explored the possibility of running load balancing in the SDN network. Their approach was based on creating as few wildcard rules as possible to insert into their switches. That enabled them to load balance proactively without involving the controller. However, they must make wildcard rules for the whole IP range and then split the IP ranges across multiple servers. The drawback of this approach is that the traffic from the whole IP range is often not uniform. They therefore had to implement an updating algorithm that changed the sizes of the IP range slices over time, which struggles with sudden changes in incoming traffic.

1.4.2 OpenFlow Based Load Balancing

At the University of Washington have Hardeep Uppal and Dane Brandon done a proof of concept, showing that there is possible to set up load balancing in the SDN network (Uppal & Brandon, n.d.). Their notations are that running one controller still produces a single point of failure and that their particular early generation of OpenFlow switches were very slow on rewriting packet header information.

1.4.3 Aster*x: Load-Balancing as a Network Primitive

At the Stanford University in USA, have they explored the possibility of building load balancing directly into a campus network (Handigol, Seetharaman, Flajslik, Johari, & McKeown, 2010). Their work focuses on reliving congested routes in their network, by diverting traffic across multiple links. This is done via a network of SDN enabled switches with their own controller application, Aster*x. The controller application determines the current state of the network and the servers, in order to choose the appropriate server and path to direct requests to. Using the NOX controller, they have demonstrated what appears to be a working prototype in multiple videos, but they only show a application GUI and no actual code has been published. Their work is however very interesting as they explore multiple problems that occurs when building load balancing directly into a typical campus network. The work of this research paper is built on the paper about Plug-And-Serve a similar solution presented

by some of the same authors in 2009 (Handigol, Seetharaman, Flajslik, McKeown, & Johari, 2009).

1.4.4 Load Balancing in a Campus Network using Software Defined Networking

Over at the Arizona State University, Ashkan Ghaffarinejad (student) and Violet R. Syrotiuk (professor/teacher) are attempting to create a SDN load balancing setup to compete with their dedicated hardware (Ghaffarinejad & Syrotiuk, 2014). Their requirement is that the SDN solution should cope better with the variation in their Campus network than the existing setup. Unfortunately as this proposed solution seems promising, it is not complete. Ashkan is at this time working on his thesis, and their publication offers no real value until the full dissertation has been completed and published.

Chapter 2

Background

This chapter is a study in literature with regards to what is the necessary information to know in order to understand the main part of this thesis. The covered topics will include the network model, load balancing, an explanation of SDNs and information about useful tools.

2.1 The OSI model

The OSI model is a model of the layers that are in play in what is commonly know as the Internet-as-a-Service terminology. In Figure 2.1 the OSI model is shown with information about what is contained in the different levels. Some of the levels that this thesis will depend upon will be described in more detail.

2.1.1 Layer 1

Layer 1 is the physical layer of the OSI model, which consists of cables and hubs. It does not, however, have to be an actual hub, but the device (switch/router) must have the capabilities of a hub included.

2.1.2 Layer 2

Layer 2 is the data link layer. It is responsible for multiple key tasks, but for the purpose of this thesis it is important to understand that in this layer, data intended for a specific MAC address is only forwarded to that unique MAC address. The protocol most used here is Address Resolution Protocol (ARP).

Layer	Example/Application	Device/protocol example
Application (7)	**End User Layer** Apps that interact with the information that was sent or recived	**User Applications** SMTP (email)
Presentation (6)	**Syntax Layer** Encrypt & decrypt	JPEG/GIF
Session (5)	**Sync & send to ports**	**Logical Ports** SQL/NFS/NetBIOS names
Transport (4)	**TCP** Host to Host, Flow controll	TCP/UDP
Network (3)	**Packets** ("letters", contains IP address)	**Routers** IP/ICMP
Data Link (2)	**Frames** ("envelopes", contains MAC address)	**Switch** **Bridge** **WAP** (Wireless) PPP (Point-to-Point)
Pysical (1)	**Pysical Structure** Cables, hubs	**Hub**

Figure 2.1: OSI-model (Zito, 2013)

2.1.3 Layer 3

As layer 2 was responsible for data to the correct MAC address, layer 3 is responsible for packet forwarding and routing through connected routers. This means that it depends on layer 2 working correctly. Common protocols at this layer are: IPv4/6 and ICMP.

2.1.4 Layer 4

Layer 4 provides end-to-end communication over multiple instances of layer 1-3 networks. TCP is the most used protocol at this layer.

2.1.5 Layer 7

Layer 7 contains all user application communication, like HTTP and SMTP (email).

2.2 Traditional Networks

Today's network devices are running a very specific set of hardware and software. From providers like Cisco that packages both the hardware and software into one single package. These devices are typically configured one by one, because every device is in their own closed environment.

This means every device contains their own configuration and flows. This makes configuration complex and it is therefore hard to achieve a good network. This also means that when there are changes to the network, upgrading the devices to accommodate the increasing load is expensive. This approach has typically been in conflict with business requirements. That is it being to static, as it does not allow any interaction with the upper layers of the OSI model. It also makes the networks very complex, meaning that to upgrade a system with a new application, may involve rebuilding parts of the data center to accommodate the innovation (Morin, 2014). The main point to take is that changing the traditional network structure is time consuming, so that when the change is in place the business decision to change the network may be out-dated.

2.2.1 Networking devices

There are many different types of networking devices, some are simple with little logic while others are complex devices with capabilities for thorough decision making. There is also mixes of them, but in this section only the three most related to network routing are shown.

2.2.2 Hub

A network hub is often called a repeater and does not manage any of the traffic. This means that for any packet coming through, it sends it out on every other port, except for the port it originated from (Pacchiano, 2006). A hub works only in the Layer 1 of the OSI model, so it's repeating function cause packet collisions which affect the network capacity (Contemporary Control Systems, 2002).

2.2.3 Switch

A network switch is a upgrade of a network hub. It sill has to have the network layer level 1, but it is created virtually in the switch. In contrast to the hub a switch only sends out a packet on one port, the correct one. There are two types of switches, unmanaged and managed switches. The latter supports configuration as the unmanaged is plug and play (CISCO, 2015). In this thesis a switch will be referred to as a managed switch. To only send out packets on the correct port, a switch keeps a record of the MAC addresses of all the devices it is directly connected to.

2.2.4 Router

A router's job is different from a switch. As the router is typically used to connect at least two networks like Local Area Networks (LANs) or Wide

Area Networks (WANs). A router is placed in the network as gateways, which is where two networks meet. Using packet header information and forwarding tables, a router can determine the best path to forward a packet (Pacchiano, 2006). Compared to a unmanaged switch, all routers are configurable in the sense that they use protocols like IMCP to communicate with other routers in order to figure out the best route between two hosts (Pacchiano, 2006).

2.3 Load balancer

The job of a load balancer is to distribute workloads across multiple computing resources. This is for optimizing the resource usage, maximizing throughput and lowering the response time to avoid overloading any single resource. Load balancers can be implemented in software or hardware often depending on what type of load they are distributing. Software load balancers are more cost efficient than hardware balancers, as they do not require specialized hardware.

A load balancer works by selecting a resource from a list of all available resources. When it receives a request it then forwards the request to the resource it chose. It may have additional functions, like maintaining the resource list by testing if the resource is up, or if the resource is at an acceptable load level before forwarding any requests to it. This is often a desired feature so that requests will be responded to by an available resource.

There are multiple methods of operations a load balancer can function. For instance the selection of the resources can be randomly chosen from a list, but that may not be the most efficient method. Another mode is *Round Robin* where it starts at the top of the resource list, and for each requests it selects the next one until it is at the end where it jumps back to the beginning to repeat the procedure. Other modes exist, for instance the *least-connections*, where it selects the resource with the least active connections.

In Figure 2.2 there is an example of how an layer 7 load balancer works. Any given client is matched to a working webserver decided by the load balancer.

2.4 Hardware load balancing

A hardware load-balancing device (HLD), also known as a layer 4-7, is a physical unit that forwards requests to individual servers in a network (Rouse, 2007). Which server it forwards to are determined by factors such as server CPU usage, the number of connections or the servers performance. The benefit of using a HDL device is that it minimizes the

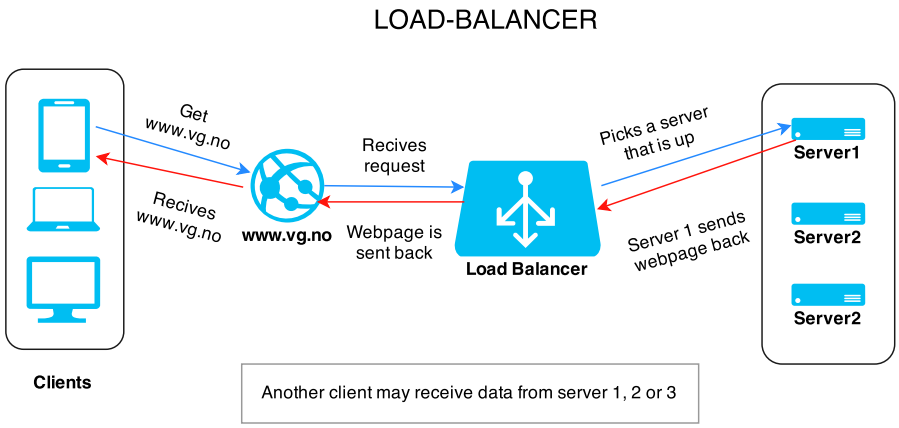


Figure 2.2: Load Balancer example for web-traffic balancing

probability that any particular resource or server in the network will be overwhelmed. In addition, it can help protect against harmful activity such as denial-of-service (DoS) attacks.

In a normal setup the HDL unit acts as a reverse proxy in order to increase capacity (simultaneously users) and reliability of applications (F5, 2015). As mentioned earlier, there are two groups of HDL devices: Up to Layer-4 and Layer-7, from the OSI model. Where as the layer-4 devices balance data in the network and transport layer protocols such as IP, TCP and UDP. The layer-7 units works in the application layer, balancing requests in for example the HTTP protocol (F5, 2015). But also based on other application specific message data, such as the spesific headers in HTTP, cookies or any other parameter that can be matched on (F5, 2015).

Companies that sell load balancers often call their HDL devices for Local Traffic Managers (LTM), This is because they manage the local traffic, as they ship with more features than just direct load balancing. This is possible because the traffic "flows" through the LTM device. One of the well-known companies besides CISCO is F5 and their product BIG-IP 1600 is explained in the next section.

2.4.1 F5 hardware load balancer

According to their sales pitch, the LTM from F5, BIG-IP 1600 series is a powerful traffic management system for enterprises designed for high performance at an affordable cost. (Armor, n.d.). It provides intelligent load balancing in addition to advanced application security, acceleration and optimization (Armor, n.d.). It's a powerful solution for improving application performance and increasing the capacity of existing infrastructure. BIG-IP LTM is a device used to reduce traffic volumes and minimize the effect of client connection bottlenecks as well as Internet latency.

The BIG-IP 1600 series security feature is described as a firewall that block attacks while still serving legitimate users. It provides network and protocol level security for filtering application attacks. Placing the BIG-IP LTM is recommended at a critical gateway to the most valuable resources in the network (Armor, n.d.). Another selling point for the BIG-IP 1600 series is that it removes single points of failure by dynamic and static load balancing methods. Some of the available balancing methods are dynamic ratio, least connections, observed load balancing and round robin.

The BIG-IP 1600 series is specified to work as a load balancer of any TCP/IP operating systems like Windows 32-64 bit, Unix based platforms and Mac OS (F5, n.d.). The specifications of the device is listed in Table 2.1.

Info	Specification
CPU	Intel(R) Pentium(R) Dual CPU E2160 @ 1.80GHz
Cores	2
Connector Type	RJ-45
Ethernet ports	4
Speed	1 Gbps
RAM	4GB
Storage Capacity	160GB

Table 2.1: Specifications for the BIG-IP 1600 series

Some use full commands for the BIG-IP are listed below: Do note that the LTM device uses one *TMM* process for each CPU. This means that the 1600-series has two *TMM* processes.

- **tmstat cpu**: Show CPU usage in real-time
- **In the tmsh shell**:
  - **show /sys tmm-info**: Show CPU usage for the last 1 second, 1 minute and 5 minutes. In addition to used memory.

2.5 SDN load balancing

There has typically been two approaches to SDN load balancing, proactive and reactive (Richard Wang, 2011) (Uppal & Brandon, n.d.). The first type of load balancing introduced with SDN was proactive and it involved dividing up the Internet address space into different slices. From here a specific slice ie. 0.x.x.x-50.x.x.x is sent to one specific server, and 51.x.x.x-100.x.x.x to another. For weak hardware switches that support a low amount of flow entries, this has been a capable solution and there has been experiments which involved changing the slices over time to accommodate load changes. There has however been some problems with this approach in situations with fluctuation with the incoming traffic as it takes time for

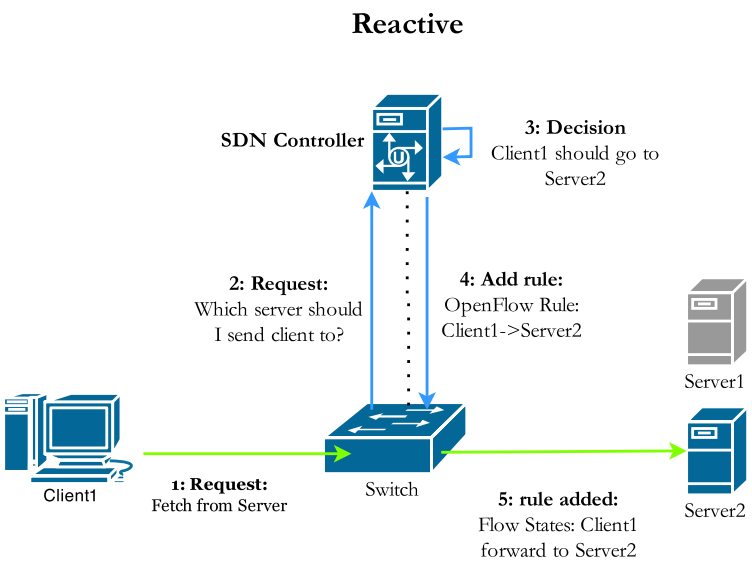


Figure 2.3: SDN Load Balancing in a OpenFlow network using a Reactive approach.

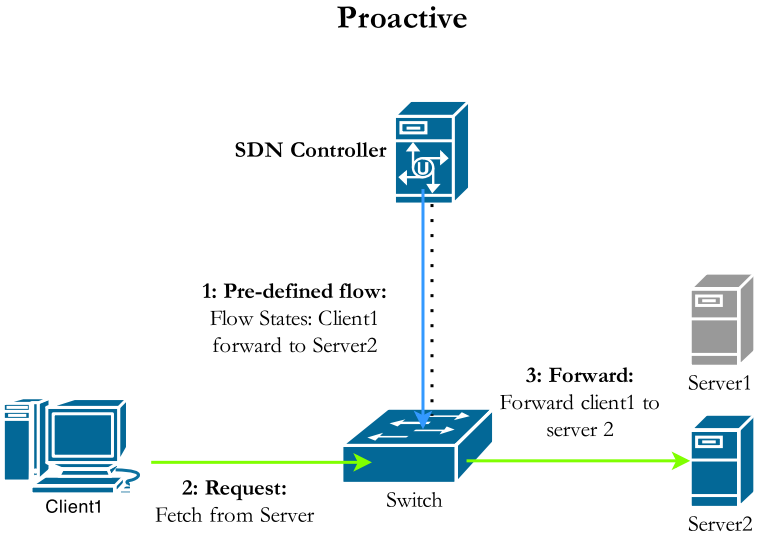


Figure 2.4: SDN Load Balancing in a OpenFlow network using a Proactive approach.

the controller to reprogram the switches to handle the changed patterns (Richard Wang, 2011).

The second approach, which is reactive have involved more load on the controller unit as it will decide which server the traffic is forwarded to

from a request to request basis (case to case basis). This works by the switches send the first packets to the controller, and then the controller programs the switches by updating their flow table to handle the rest of the flow. The problem with this approach is that it can overload the controller when the network load exceeds the processing power. It also leads to longer processing time in the switch for new requests. This means that for short burst of load, reactive balancing may be too slow to be feasible, but for longer flows letting SDN run load balancing can be beneficial (Phaal, 2013).

The difference between these modes of operation are explained in figures 2.3 and 2.4. They show that in a reactive environment nothing happens until a packet enters the switch, which forwards the packet to the controller because it does not know what to do with it. From here the SDN controller has to decide what to do with the packet before installing a flow in the switch. Now that the switch has a flow rule of what to do with client1's packets, it forwards all incoming packets from client1 to server 2.

In the proactive mode of operation the SDN controller programs the flow rules into the switch before any packets has been received. This way when client1 sends a packet the switch just forwards the packets to server 2, according to the flow rule.

2.6 Software Defined Networking (SDN)

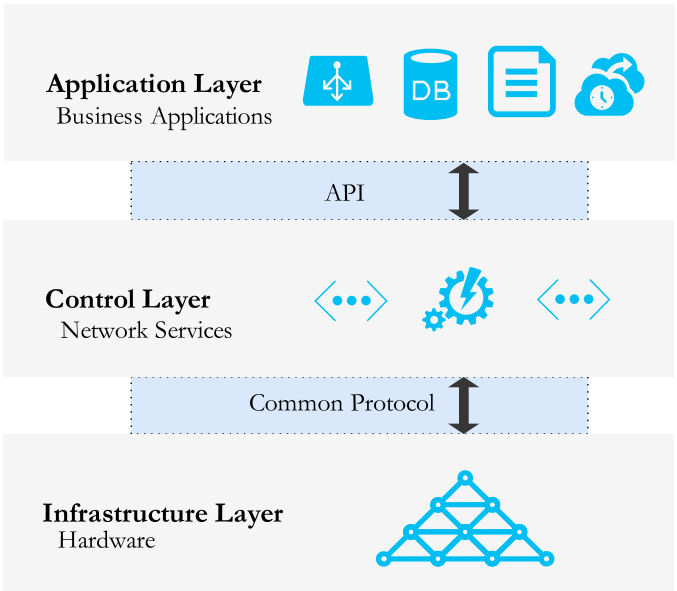


Figure 2.5: SDN model (open networking foundation, 2015)

Software Defined Networking is a new approach to networking where the control plane is decoupled from the data plane (Feamster, Rexford, & Zegura, 2013). The control plane is the system that makes decisions

about where traffic is sent, where as the data plane is the underlying systems that forward traffic to the decided destination. Because SDN offers a logically centralized controller, it lets administrators dynamically adjust the traffic flow for the whole network to accommodate changing needs. (open networking foundation, 2015)

Services that can run in the network OS of SDN:

- Routing
- Multicast
- Security
- Access control
- Bandwidth management
- Traffic engineering
- QoS
- Energy efficiency
- Other types of policy management

(Azodolmolky, 2013)

Interaction between the planes, from software to hardware is done via a Common Protocol where as OpenFlow is the most famous and talked-about southbound protocol.

As detailed in Figure 2.5 the structure of SDN consists of three layers. Which are explained in the next sections.

2.6.1 Application Layer

This layer consists of applications that runs on top of the network, like a load balancer, firewall or an application that runs loop avoidance (Edelman, 2013). Its interaction with the control level should be done via an API, often referred to as the northbound API. However, the northbound API does not have well-defined standard as the southbound protocol has. But the intent with a well defined northbound API is to create something like a "app store" for networking (Johnson, 2012).

2.6.2 Control Layer

This layer consists of applications that run the network, like network routing. In other words; The network operating system (Feamster et al., 2013). In this layer there is an SDN controller that decides the flow of

packets in the network. Many controllers exist, but they all follow the same principle. The interaction with hardware is often refereed to as the southbound protocol, because it is the layer below. This makes the control layer the glue that binds the planes together. The most widespread and talked about southbound protocol is OpenFlow. There are vendor specific alternatives to OpenFlow, but they all are common protocols for interaction between the layers.

2.6.3 Infrastructure Layer

This layer consists of the actual physical/virtual hardware, like a switch. This hardware is not specialised for the tasks the above layers demands, it can be just a simple switch that supports SDN and packet forwarding. Everything logical has already been taken care of by the controller, so the devices in this layer only follows orders (Feamster et al., 2013).

2.7 OpenFlow

OpenFlow was the first standard interface between the control and data plan layers in networking. It aims to ease the work for the network administrators by implementing a SDN open common protocol (Feamster et al., 2013).

For a scenario where a data center has thousands or even hundreds of servers connected to the network, managing separation like VLANs (virtual LAN) on every closed box in the network would be an enormous task. Adding that many networks dynamically change, it leads to problems for the network administrators. What SDN can do by using the OpenFlow standard is to centralize this task to a logical controlling unit, where it is easy to program VLAN like functionality. Do note that OpenFlow in it self does not provide the standard VLANs, but rather a Layer3 policy of who can talk to whom (Cole, 2013), (Feamster et al., 2013).

A visualisation of how OpenFlow differs from a traditional network setup can be seen in Figure 2.6. Here it shows that the traditional network environment has the control and data plane contained in each unit, where as in the OpenFlow enabled environment it is separated.

OpenFlow is just a protocol, namely a specification on how the communication between the control and data plane is handled. It runs over TCP and has support for SSL to secure the communication between the switch and controller.

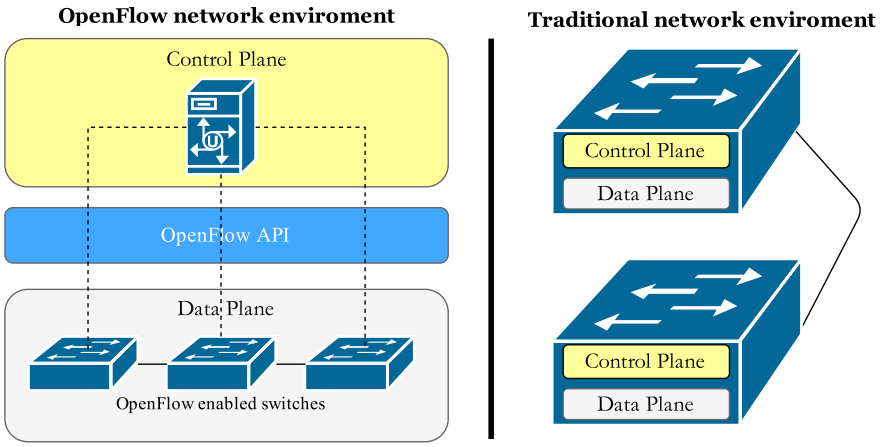


Figure 2.6: OpenFlow enabled network vs traditional

2.7.1 Network Flows

Flows is simply stated how objects move through a network. In SDN the network flows are basically packets as a object consists of multiple packets in most cases. Point-to-point communication is a example of what would be a network flow, as there may be multiple exchanges but they are tied together by the characteristics listed below. There is different type of methods to classify a flow, but without looking at the content we can aggregate packets into a flow based on these characteristics from the transport layer header (Asai, Fukuda, & Esaki, 2011) (Feamster et al., 2013):

- Packets must have the same:
  - Protocol
  - Source IP and port
  - Destination IP and port

Packets should also have have happened in some defined amount of time to be classified together. Do note that TCP and UPD packets appear as two and one flows respectively with this classification. Because UDP is unidirectional (moving in one direction) and TCP is bidirectional (moving in two/both directions) (Asai et al., 2011).

Network flows are the cornerstone of how OpenFlow functions as a switch, because the OpenFlow *flow table* consists of rules intended to match a specific flow.

2.7.2 OpenFlow Flow Table

There may be multiple Flow tables in a switch, like firewall, QoS, NAT and Forwarding. But this section covers what is in one table. In the following Table 2.2 is an very short example rule set for a Flow table: Do note that this is only a simplified example and that the actual table is more complex.

Table 2.2: OpenFlow table example

#	**Header Fields**	**Actions**	**Priorities**
1	if in_port = 1	output to port 2	100
2	if IP = 10.1.2.3	rewrite to 84.4.3.2, output port 6	200

The first rule in Table 2.2 states that if a packet arrives into the switch on port 1 it should be sent out on port 2. But if a packet arrives at port 1 and it has source IP 10.1.2.3 there is suddenly two rule matches for that particular packet. To then decide what will happen to the packet the column *priorities* will be used. In this case as the second rule has priority of 200 which is greater than 100 for the first rule. The switch would then use rule 2 and rewrite the IP to 84.4.3.2 and output it on port 6.

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Protocol	TCP sport	TCP dport

Figure 2.7: OpenFlow match fields for packets.

Figure 2.7 shows the fields that OpenFlow can match packets against according to the OpenFlow v1.0 specification (OpenFlow, 2009).

Table 2.3: OpenFlow entries columns (OpenFlow, 2013).

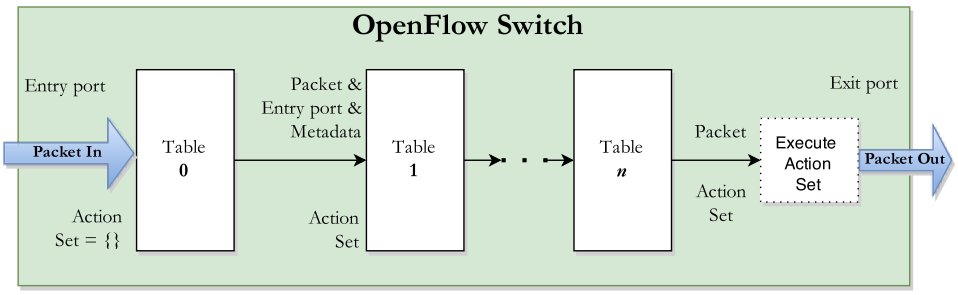
Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

A flow table consists of multiple flow entries, or rules as exemplified in Table 2.2. According to the OpenFlow specification 1.3.3 (OpenFlow, 2013), these fields as shown in Table 2.3 are the columns that make up an entry in the Flow table.

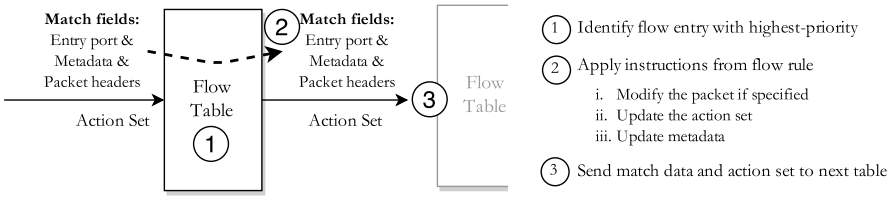
Tying it all together; The flow entries make up one flow table, and multiple flow tables make up the processing pipeline of OpenFlow which is explained next.

2.7.3 OpenFlow PipeLine

Every OpenFlow switch is required to have at least one flow table but they can contain more if needed. The pipeline of an OpenFlow switch defines how packets interact with the flow tables, as shown in Figure 2.8. The figure



(a) A packet passing through a OpenFlow switch is matched against multiple tables in the pipeline



(b) Packet processing in each table

Figure 2.8: The pipeline of OpenFlow (OpenFlow, 2013). (a) for the whole pipeline. (b) for one table in the pipeline

assumes multiple flow tables as the pipeline with only a single flow table is greatly simplified.

In OpenFlow pipelining the flow tables are sequentially numbered, starting at 0. The processing of packets always starts at the first flow table (0), where a packet is match against the flow entries it contains. Depending on the outcome of the packet processing in the first table, they other tables may be used (OpenFlow, 2013).

Every packet in OpenFlow has an action set associated with it. This is by default empty and follows the packet through the whole pipeline process as shown in Figure 2.8. The action set can be modified by flow entries, accommodating changes until it reaches the execute action at the end of the pipeline. When a flow table processes a packet, it is matched against the flow entries of that flow table. If a matching flow entry is found, the actions set (instructions) for that flow entry is executed. A instruction may contain the use of the *GotoTable* action where the packet is sent to another table where the same process happens again. Do note that the *GotoTable* action may only direct a packet to a flow table with a lager table number than itself. This means that the pipeline processing cannot go backwards only forward. Obviously, then for the last table there cannot be any *GoToTable* instructions. If there is not any *GotoTable* instructions in a flow table that matches for a specific packet, the pipeline processing stops and executes the action set it has acquired so far (OpenFlow, 2013).

On the other hand, if a packet does not match any flow entries in a flow table it is a *table miss*. What a switch should do with missed packets

depends on the configuration in the form of a *table miss flow entry*. These options for the switch are to drop the packet, pass them to another table or send them to the controller (OpenFlow, 2013).

2.7.4 OpenFlow versions

The first version of OpenFlow was released in 2009. It would then be almost two years before version 1.1 came out and added support for Multi-table pipeline processing, MPLS and QinQ. Followed by the release of version 1.2 10 months later, it added support for IPv6 and additional extensibility. In 2012 version 1.3 was released adding support of QOS alongside with other features, followed by the release of 1.4 in 2013 (Oliver, 2014). Version 1.4 introduced support for decision hierarchy and multiple controllers along with other features. At the end of last year, 2014, the specifications for 1.5 was released and approved by the open network foundation (ONF) board, but has not yet been approved by all parties and finalized.

Although new releases of OpenFlow come out, there is a lack of vendors including support for the newest versions of OpenFlow in their products before the marked demands it (Oliver, 2014).

2.8 OpenFlow switch

Because a switch that runs the OpenFlow protocol has it's decision making taking away from it, it differs from a normal switch. In a way this simplifies the switches as they don't think for them self, but rather have their decisions taken by a central controller. (Limoncelli, 2012). Every OpenFlow switch has to have the support of at least one flow table, which is changed by the controller via add, delete or update. A flow table contains flows; namely a rule for a specific packet occurrence. These flow rules specify what to do with a packet if it matches the criteria of the flow rule.

These flows can be installed in the switch proactively by installing them before any packet comes in, or reactively where as when the switch receives a packet without a matching flow it asks the controller for what to do with the packet.

Not all switches support this protocol, but it is getting more common that the switch vendors are including OpenFlow support in their products. Because including support has often been a simple firmware upgrade.

The biggest constraint with OpenFlow switches has been in the lower-end section as they come with less TCAM space. TCAM is a special lookup RAM for switches that can take three different inputs: 0, 1 and X (Salisbury, 2012). It is used to store flow rules, for fast lookups. But as OpenFlow

support very fine grained control, there may be many rules required and running out of TCAM space to store the rules is a problem.

2.9 OpenFlow controllers

There is a wide selection of OpenFlow controllers, like NOX, POX, Trema, Beacon, OpenDaylight and Floodlight (Pronschinske, 2013). However in this thesis the controllers POX and Floodlight has been chosen.

2.9.1 POX

POX is a framework for interacting with OpenFlow switches written in Python (NOXRepo.org, 2015). It is a sibling of the original SDN controller NOX (written in c++), where their main difference is their scripting language. Due to POX being written in Python it is the recommended controller to start with according to Murphy McCauley, the maintainer of NOX/POX (Chua, 2012). This also meas that it can be run under most platforms such as Linux/Unix and Windows. A POX installation includes different modules that resembles different type of normal switch behavior, in addition to other routing modules it supports custom modules. Some of the POX components provide core functionality, some are for convenient features and some are just examples. In the list below some of these components are listed (M. McCauley & Al-Shabibi, 2014):

- **forwarding.l2_learning:** This POX component enables a OpenFlow switch to become a layer 2 (L2) learning switch. This component tries to make flow rules are exact as possible. I.E, it tries to match on as many fields as possible. This means that different TCP connections will results in different flow rules.

- **forwarding.l3_learning:** This component should be a router as it is labeled as an L3 device, but it is not quite a router. It is a *L3-learning-switchy-thing* (M. McCauley & Al-Shabibi, 2014), and used to test ARP requests and responses.

- **forwarding.l2_multi:** This component is sort of a learning switch, but is has an additional feature. Normal learning switches learn their connections on a switch-by switch basis, making decisions only about what they are directly connected to. l2_multi uses the *openflow.discovery* and learns the topology of the entire network. This means that when one switch learns where a a MAC address is, they all do and can therefore make decisions based on that.

- **openflow.discovery:** This component uses the Link Layer Discovery Protocol to discover the network topology.

- **openflow.spanning_tree:** This component uses the information from the *openflow.discovery* component to provide a loop free network. It works sort of a Spanning Tree protocol, but it is not the exact same thing.

- **openflow.keepalive:** Because some OpenFlow switches assumes that an idle connection to the controller is the same as loss of connectivity and will disconnect after some time. This component ensures that the connection is refreshed by periodically sending ECHO requests, so they switches does not disconnect.

- **proto.dhcpd** This component acts as a DHCP server, leasing out DHCP addresses to clients.

- **misc.gephi_topo:** This component provides a visualization of switches, links, and detected hosts.

2.9.2 Floodlight

Floodlight is an OpenFlow controller written in Java that requires few dependencies, enabling it to be run on a variety of platforms. Released with the Apache license, Floodlight can be used for almost any purpose (Floodlight, 2015).

Similar to POX, it is also module based, making its functionality easy to extend. In addition, Floodlight delivers high performance, as it is the core of a commercial product from Big Switch Networks. It comes with support for many different virtual and physical OpenFlow enabled switches. Interactions with the controller are issued using the *REST API*, which is an API that uses the HTTP protocol for easy interaction with the controller. (Floodlight, 2015).

As of version 1.0, support for OpenFlow protocol 1.0 and 1.3 are stably implemented, any versions of Floodlight before 1.0 only supports OpenFlow 1.0. Other versions of OpenFlow only have experimental support in Floodlight (Floodlight, 2015).

Some Northbound API applications come bundled with Floodligh, these are OpenStack Quantum, Virtual Switch, ACL (stateless FW) and Circuit Pusher. But other applications can be written and loaded as modules.

Because Floodlight uses an HTTP based API it also has a GUI that is accessible via a web browser as shown in Figure 2.9.

2.10 Testbeds

As OpenFlow enabled devices are not common hardware at the University of Oslo, working with SDN networking will require a virtual environment

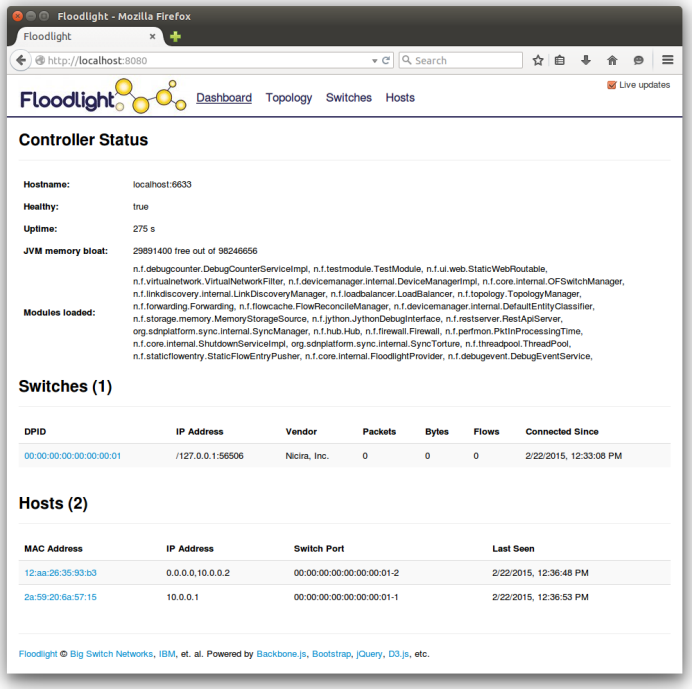


Figure 2.9: Floodlight GUI at start page

for testing purposes.

2.10.1 Mininet

Mininet is a application that can create a network of virtual hosts, switches, controllers, and links on a single machine. Spawned switches support OpenFlow for Software-Defined networking emulation and the virtual hosts run standard Linux software. As this closely emulates a physical network, it is ideal for research proposes. It relies on a Linux feature called network namespaces, making kernels above version 2.2.26 a requirement for Mininet to function. But this also means that it runs real code including standard network applications as well as the real Linux kernel and network stack. As it supports arbitrary custom topologies, any custom network can be emulated. To be able to setup a custom network, Mininet has a Python API that allows the creation and testing of networks via a python script. (Team, 2015)

Mininet has been mostly used to demonstrate proof of concept's, instead of performance because there is a overhead when emulating data flows. This is because packets first need to be exchanged between the virtual switches to emulate packet flows. Which results in the switch sending a Packet-In to the controller, where a kernel to user space context switch happens and induces overhead. This slows down the control plane traffic the Mininet

testbed can emulate.

The advantages for using Mininet for research purposes as compared to a fully deployed virtual network is that is uses less resources, boots up faster, scales better and that it is easy to install. That is why Mininet is going to be used in this thesis as a testbed for running OpenFlow controller against Open vSwitches. vSwitch is referred to *ovs* in the Mininet *cli*, and is a open source, production quality, multilayer virtual switch.

2.11 Benchmarking/assisting tools

2.11.1 Iperf

Iperf is a command line utility to measure bandwidth between hosts. In order to use it one of two hosts must be started as a server and the other as a client.*Iperf* works by setting up a TCP or UDP connection where it pushes as many packets between the client and server as it can, and measuring the bandwidth it achieved between them. (Iperf, 2014) Both software's parts are bundled in the same repository package, and the mode of operation is selected at boot for either client or server.

2.11.2 Wireshark

Wireshark is a free and open-source application for packet analysis and network troubleshooting. It is a cross-platform application, running on most Unix like operating systems and Microsoft Windows (Wireshark, 2015). It has support for dissecting most networking protocols, but some protocols requires a dissector plugin for optimal usage. It will capture packets that are unknown, but to decode specific information about the OpenFlow packets or to filter them correctly the openflow.lua plugin is needed.

By providing the user with a GUI (Graphical User Interface), Wireshark allows live view of the network traffic on the network card it is listening to. In this thesis Wireshark will be used to listen at the loopback interface, as all the Mininet testbed traffic passes through that virtual network card.

2.11.3 Tcpdump

Tcpdump is very similar to Wireshark, as this tool also captures packets passing an interface. There is, however, no GUI as it is a command line tool. It runs on most Linux systems and will be used on the individual virtual hosts in Mininet to analyze network behavior.

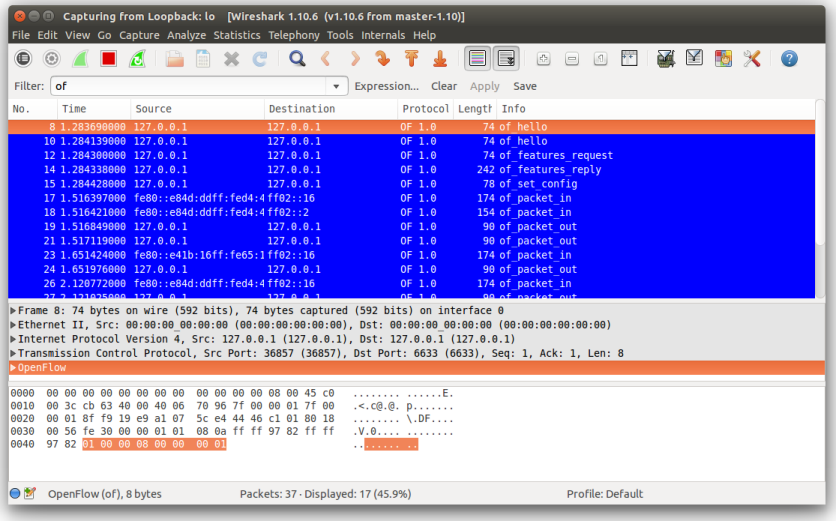


Figure 2.10: WireShark GUI with OpenFlow traffic packets captured

2.11.4 Oracle VM VirtualBox

VirtualBox is a x86 virtualization software developed by Sun Microsystems that is freely available as Open Source. It allows the use of fully functional operating systems to run virtually on a host system.

For this thesis VirtualBox will be used to run the Mininet image on a OSX host computer.

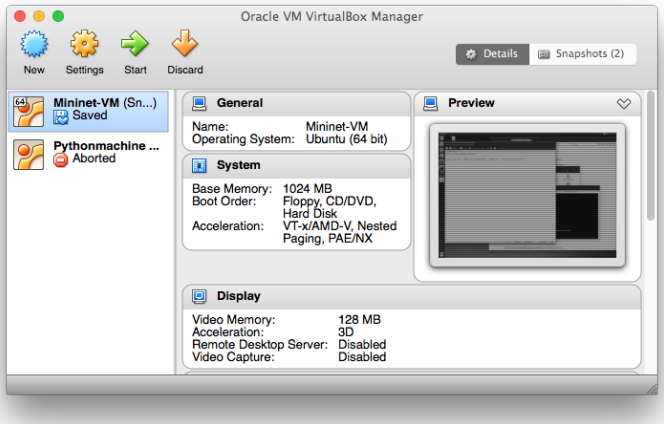


Figure 2.11: VirtualBox GUI for Mininet instalation

2.11.5 ethstats

This tool is useful for the Mininet installation, as it allows for quick overview of Ethernet statistics for the whole network.

2.11.6 httperf

httperf is a tool to measure web server performance over the HTTP protocol (Laboratories, n.d.-b). The supported protocols are HTTP/1.0 and HTTP/1.1, bundled with a variety of workload generators (Laboratories, n.d.-b). It functions by imitating a number of clients accessing a given website, which induces load on the server. Because all request are run from the same program it can measure the servers responses. Of of it's ways to measure a server is to generate a provided number of HTTP GET requests and then measure the feedback from the server (Laboratories, n.d.-b). What to look for in this feedback is then how many of the sent GET requests that are responded to, and at what rate the responds come back at (Laboratories, n.d.-b).

A normal way to run httperf is in a client/server relationship, this means that it can be used to benchmark any type of webserver that uses the HTTP/1.0-1 protocol(Laboratories, n.d.-a). There are multiple options available when running httperf, but these options must be set in order to run a successful test. These options are not specific for, but holds true for webserver benchmarking.

- How many connections to make each second.
- How many of these connections should it make in total
- How many requests should be made for each of those connections
- What is the "timeout" limit. I.E. how long to wait, before considering the request not completed within a satisfactory amount of time.

Of these options the rate will be the most important, but without setting the others it is not possible to achieve a correct test.

Below is an explanation of the outputted information from httperf:

- **Num-conns**: The total number of requests to send.
- **Request rate asked**: The rate of which to try to send requests.
- **Completed rate**: The actual rate of requests received/completed.
- **Duration**: How long the test took to complete.

2.11.7 ApacheBench (ab) - Apache HTTP server benchmarking tool

ab is another tool to measure web server performance over the HTTP protocol.

The ab output and options are explained below:

- **RPS**: Requests Per Second, how many requests per second achieved.
- **Duration**: How long the test took to complete.
- **Kbytes/s**: The transfer rate achieved in Kbytes/s.
- **ms**: Mean time per requests (ms), for all concurrent requests (individually)
- **-c**: How many concurrent requests to send.
- **-n**: The total number of requests to send.

Part II

The project

Chapter 3

Planning the project

This chapter is dedicated to the planning of the project, often referred to as the approach. It gives an overview of what to come, by introducing the environments and methodology. In addition to this, it provides an insight into how the experiments will be conducted. The main idea of this thesis is to evaluate how feasible an SDN-based load balancer is. In order to do so, it must be compared to an existing solution. A good reference point would then be a hardware load balancer, tested against the SDN-based solution in terms of performance. But as there are multiple SDN controllers available, testing two different controller solutions to load balancing is beneficial in determining how viable SDN-based load balancing is. However, existing solutions may not bee fast enough, so that developing the fastest possible scenario should give reference data for determining SDN-based load balancing feasibility.

3.1 Testbed design

In order to compare a result there is a need of at least two different input parameters. In this case the inputs will come from different setups, but as there is a lack of available OpenFlow enabled switches at the HiOA/UiO campus for experiments, one of the setups must be virtualized. As for the other setup HiOA provides Jottacloud's old F5 hardware load balancers, with corresponding server hardware.

The next sections will explore the different designs as we try to match the virtual environment to our hardware environment as closely as possible.

3.1.1 Virtual environment with Mininet

For research purposes regarding SDN, Mininet is the obvious choice when it comes to performance and ease of use. The provided image of Mininet

is installed as a Virtual Machine in VirtualBox. The specifications for the virtual environment are shown in Table 3.1.

Type	Specification
VirtualBox version	4.3.20
Mininet version	2.2.0
Virtual OS	Ubuntu 14.04
Virtual RAM	1 GB
Virtual HDD	8 GB
Virtual NIC	1 Bridged Adapter
Amount of CPU's	1
Host OS	OS X 10.10
Switch Type	OpenVswitch

Table 3.1: Specifications for the virtual environment

Figure 3.1 gives an overview of the virtual environment. This figure shows that in a virtual environment the network administrator or in this case the researcher, only interacts with one virtual instance on a host computer.

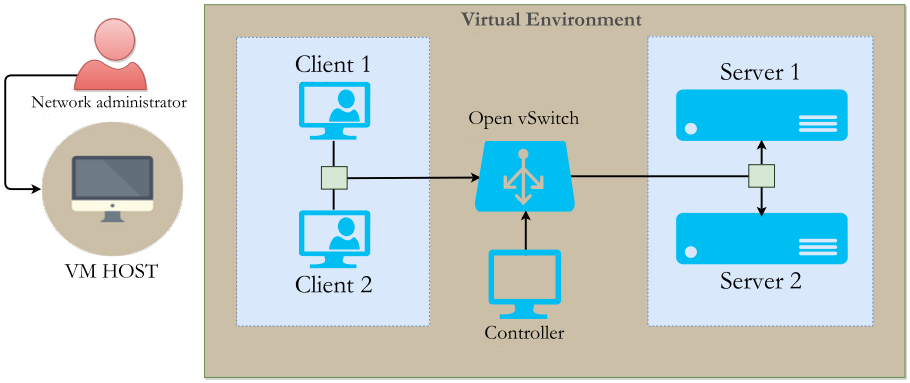


Figure 3.1: Testbed virtual setup

To start the virtual Mininet environment with a functioning minimal topology, this command should be issued:

Start Mininet

```
sudo mn
```

To run Mininet with an external controller (POX/Floodlight) as is needed for our tests, these parameters should be included:

External controller options in Mininet

```
sudo mn --controller=remote,ip=127.0.0.1,port=6633
```

3.1.1.1 Limitations and solutions of the virtual environment

Even thought running on a single system is convenient, it imposes resource limits. For instance on a 3 GHz CPU there is possible to have about 10 Gbps of simulated traffic, shared between the virtual hosts and switches (Team, 2014). That should however be enough bandwidth to simulate the hardware environment. One important factor will then be to utilize the link limit parameter in Mininiet. This is because there is no bandwidth cap by default, so instead of running at a physical cap of 1 Gbps as in our physical environment, Mininet will use what it has available. If the virtual environment is experimented on without any limits, it may produce unwanted results when compared to a environment with fixed limits as the the hardware environment have. This is why all links should have a defined speed using Mininet commands. Defining a speed and minimal set up topology in Mininet is done via the command line by passing a parameter to the Mininet startup command like this: *–link tc,bw=10,delay=10ms*

The possible link parameters are listed in this list (Team, 2014):

- **bw** : Defined as Mbit. (1 Gbps is around 125 Mb/s or 1000 Mbit/s)
- **delay** : Delay is defined as a string with units in place (e.g. '5ms', '100us', '1s')
- **max_queue_size** : The maximum amount of packets in the queue
- **loss** : Loss of packets is expressed as a percentage between 0 and 100.
- **use_htb** : Hierarchical Token Bucket rate limiter, True or False.

3.1.1.2 CPU limitations in Mininet, and resource allocation problems

In every environment there is resource pool, meaning that there is a limit of the amount of resources available. For a typical physical environment the limitations are fixed, meaning that a server has some amount of resources allocated to it. It may then use all or just a fraction of its available resources, without affecting the other resources in the pool. For a typical virtual environment this configuration imposes some problems with a load balancing scheme. Because in contrast to the physical environment where the resources are separated, the virtual environment must share the same resources. Sharing the same resources means that the CPU is allocated to all tasks the same amount. This is the part where the problems start, because when adding more web-servers to the setup you only increase the complexity without adding any resources. As the resources stays the same

and the complexity is increased, the overall performance is slower as you add more servers, which is normally not a desired result.

To overcome this problem every virtual hosts in the Mininet environment must have a CPU limit parameter. We aim to achieve around 10 % CPU allocation for each host, this means that 1 server can throughput 10 percent of the CPU and 2 servers 20 percent. This means that when adding servers the resource allocation works like a physical setup.

The suggested parameters for host configuration are finalised in the network boot script, but below are the parameters just for CPU limit:

```
CPULimitedHost, sched='cfs', period_us=10000, cpu=.025
```

3.1.1.3 Configuring link parameters in Mininet

To closely mimic the physical network in our virtual environment, a network test measuring achievable bandwidth from the hardware is going to be conducted. This test will involve the networking tool iperf for bandwidth throughput tests, and ping for a delay test. With these results, it is possible to set the link parameters so that the virtual network is comparable to the physical one. The expected *bw* parameter is to be below, but close to *1000 Mbit/s* and the delay less than *1 ms*.

3.2 Hardware environment build

Building the hardware environment involved a more complicated setup than originally planned for because of the inner workings of the F5 load balancer. However, this more complicated setup also connected it to the Internet allowing for remote access and more controlled administration.

3.2.1 Configuration of servers and clients

The final setup as shown in Figure 3.2, shows that some additions had to be included to accommodate the policy's of the BIG-IP 1600. The main addition to the setup is the use of VLAN tagging, where as the servers and clients are on two different VLANs. A VLAN tag is a packet encapsulation mechanism of normal packets according to the IEEE 802.1Q standard. Its main functionality is to virtually separate hosts in a network from each other, for instance in a scenario where some clients are not allowed interaction with a specific server network. Because encapsulation is normally done in switches and because this setup does not have any, there was a problem with encapsulating the packets from the machines connected to F5 bridge. That is why the next section covers the steps on

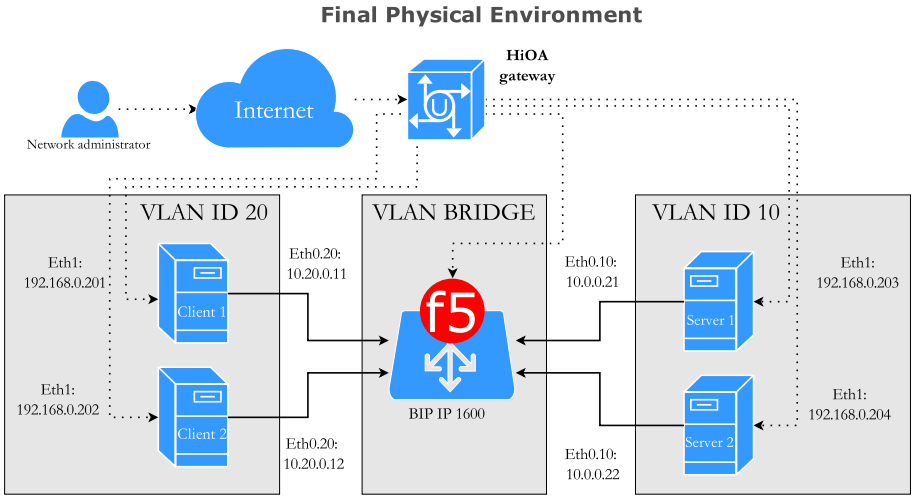


Figure 3.2: Final hardware setup.

how to configure Ubuntu 14.04 machines so that they encapsulate packets as 802.1Q.

First, before any commands for VLAN tagging can be run, the interface file has to be configured for VLAN tagging with it's VLAN ID. Below is an example of how the interface file looks like on client 1:

/etc/networking/interface content for VLAN enabled NIC

```
auto eth0.20                    # 0.20 -> VLAN id = 20 on eth0
   iface eth0.20 inet static    # eth0.20 to static IP
      address 10.20.0.11        # address of eth0.20
      netmask 255.255.255.0     # netmask /24
      vlan-raw-device eth0      # what physical device eth0.20 is for
```

Installing a VLAN tag for a specific NIC is shown in listings 3.1, which shows the specific commands needed to setup client 1 in the network after the networking file have been updated. The commands for the other servers are identical except for the IP address and VLAN tags that are changed appropriately.

Listing 3.1: Commands for configuring VLAN trunk tags on a Ethernet interface in Ubuntu 14.04 for client 1

```
1 # Run commands as root:
2 $ echo "8021q" >> /etc/modules      # Enable VLAN tagging at boot
3 $ modprobe 8021q                    # Enable VLAN tagging at runtime
4 $ apt-get install vlan              # Needed for vconfig command
5 $ ifconfig eth0 inet 0.0.0.0        # Remove route from eth0
6 $ vconfig add eth0 20               # Add VLAN tag 20 to eth0
7 $ ifconfig eth0 up                  # Set network status to up
8 $ ifconfig eth0.20 up               # Set network status to up
9 $ ifconfig eth0.20 inet 10.0.0.11   # Set static IP
```

3.2.2 GUI configuration of BIG-IP 1600

This section covers the parameters that were needed to configure the BIG-IP load balancer for load balancing and with proper routing. The BIG-IP has one management (MGMT) port and four ports, which are numbered 1.1, 1.2, 1.3 and 1.4. Where as the MGMT port is separated from the server ports, so that no traffic can cross between them. This means that for the following configuration, some steps are configured after which port the machines is connected to. Which machine that is installed into which ports are as follows:

- 1.1: Client 1
- 1.2: Client 2
- 1.3: Server 1
- 1.4: Server 2

Which clients and servers that are installed on which ports are important in regards to the VLAN setup of the BIG-IP. This is because they are going to be on separated VLANs and the VLANs are tagged to specific ports. The two VLANs are public and private, with the private VLAN containing servers, and the public contain the clients.

- Private, VLAN tag = 10, subnet 10.0.0.0. Ports = 1.3, 1.4
- Public, VLAN tag = 20, subnet 10.20.0.0. Ports = 1.1, 1.2

Because VLANs are a separation of networks, they need to be bridged in order to communicate together. This is done via the *"Network ›› VLANs : VLAN Groups"* menu where "create" is selected and these options are filled as shown in Figure 3.3

Finally to actually connect the BIG-IP to the network, we assign it a *self IP* at *10.0.0.50* connected to the VLAN network bridge, *bridge* as shown in Figure 3.4. The configuration with how the separation of the VLANs works and how they are connected are logically virtualized in figure 3.5. With the VLAN configured, everything is ready for the load balancing setup, which is covered next.

3.2.2.1 BIG-IP Load Balance Setup

There are two parts needed to be setup for load balancing, which both are a part of the *Local Trafic* settings of BIG-IP:

- **Virtual IP**: An IP clients queries for their requests, but not actually bound to a server.
- **Backend pool**: The servers responding to a request sent to the virtual IP.

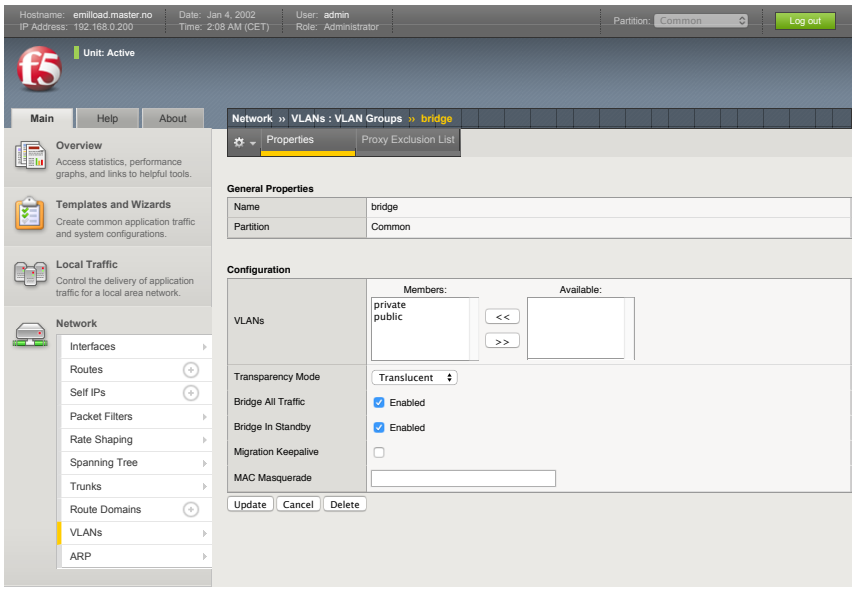


Figure 3.3: Configuration of networking bridge of the two VLANs in the F5 GUI setup

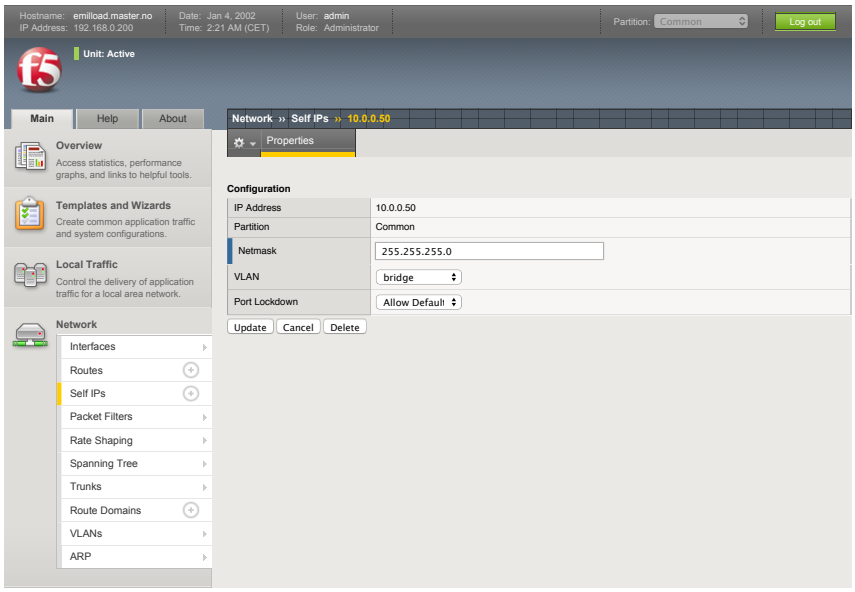


Figure 3.4: Configuration of self IP in the F5 GUI setup

The settings for the *Backend Pool* is found under the menu *"Local Traffic ›› Pools : Pool List"* which covers the backend (pool) setup. In this example, even though the name of the pool can be generic it is named *"Backend-servers"*. Here there are two parts that completes the pool, the properties of the pool and the members it have. The first settings page as shown in

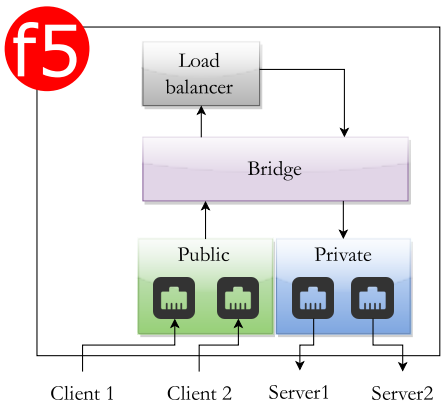


Figure 3.5: Inner workings of the F5 VLAN bridge and the Load Balancing module.

Figure 3.6 is important because the BIG-IP needs to know that the backend servers are working before it forwards any requests to them. So unless it is not configured correctly the load balancing won't function.

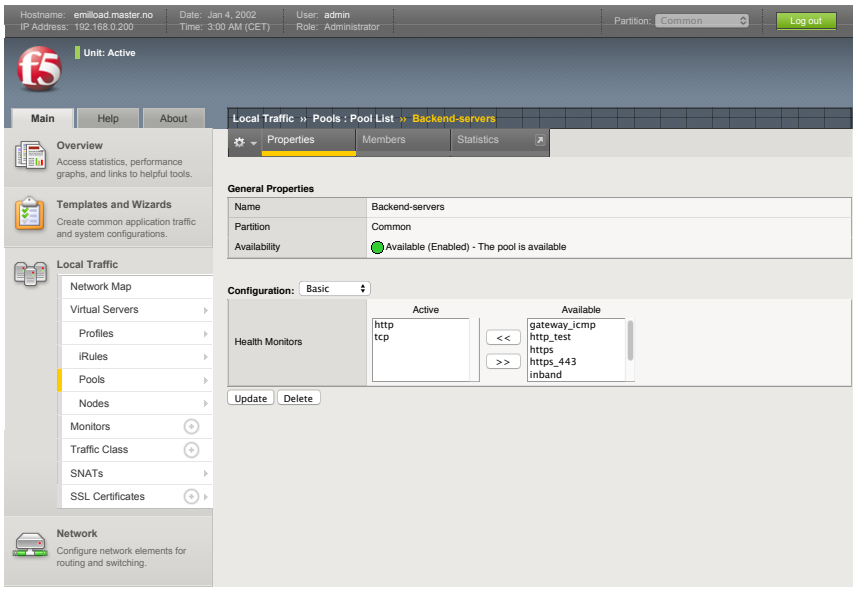


Figure 3.6: Health monitoring properties for the pool.

The second settings page is important, because as shown in Figure 3.7, the members of the pool and the load balancing algorithm to use is selected at this configuration page. Do note that there are many different types of algorithms to select, but that the *Round Robin* configuration is used for this setup.

The second most important part is the Virtual IP, for this example the IP chosen is *10.20.0.100*, and the name for it's configuration is *Virtuel-server-*

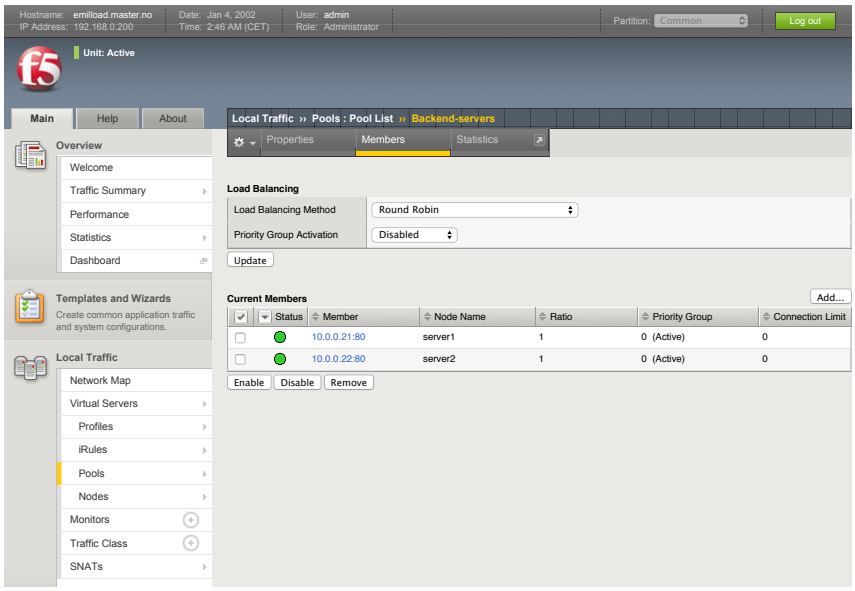


Figure 3.7: Pool members and load balancing algorithm.

100. Do note that the IP is part of the public VLAN as that is where the clients are connected, as they are the machines that will be using the Virtual IP. The complete set up for this is shown in Figure 3.8, which is the first settings page. Here it is configured for *Standard* load balancing on port *80*, but other ports or protocols can be selected. The second page "resources" is not shown, but on this page the backend *"Backend-servers"* is selected as those servers should respond to requests at this virtual IP.

3.3 Load Balancing methodology for SDN

As discussed in the background section, there have been two different approaches to SDN-based load balancing because of limitations to the different methods.

Reactive gives more fine grained control, but are limited by the processing power of the controller and slowing down every request. In contrast to proactive that gives less control, but limits the use of processing power and request completion time. Depending on the network, using both modes for different setups/services may be beneficial. As for some services it may be acceptable with higher strain on the controller if it copes with network changes better. Finding out how the two modes behave will be part of the process of determining how SDN-based load balancing functions in comparison to a hardware solution.

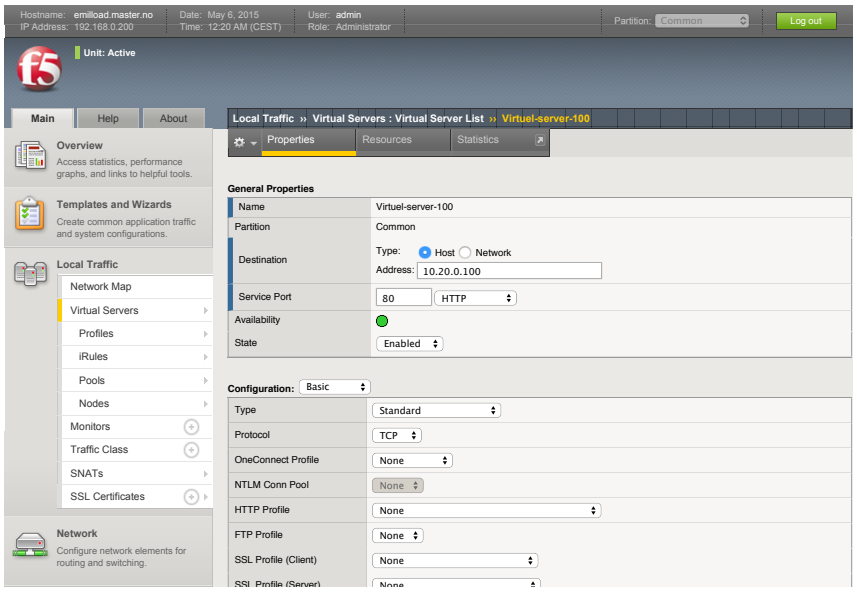


Figure 3.8: Virtual IP settings page.

3.4 Comparing SDN controller solutions

The first problem statement bases itself on finding an SDN-based solution to load balancing, but as there are multiple solutions there is no way of knowing beforehand which is the most feasible. That is why two controllers approaches to SDN load balancing be tested. Sourcing the available controllers that come with load balancing capabilities, while excluding any commercial controllers, have determined which controllers to test. From here it is clear that because the way a reactive solution works, it introduces some network delay in any SDN solution. Testing more than two controllers would therefore not be necessary unless these tests show a large variation in performance.

3.4.1 POX SDN controller

POX has been chosen over NOX as it is the latest controller for scientific research recommended by the developers of both controllers. As POX is Python-based, it officially requires Python 2.7. Newest version is the *eel* branch which is available at GitHub, but the *carp* version that ships with Mininet is updated. This means that the latest branch will need to be manual downloaded and used for tests in this thesis. The default port for a POX controller is *6633* which therefore should be included in the *mn* startup command. The startup command is described in the 3.1.1 section. POX can be started as a l2 learning switch SDN controller with the following terminal command:

POX start

```
./pox.py forwarding.l2_learning
```

Multiple modules can be specified like this:

POX multiple modules

```
./pox.py samples.pretty_log forwarding.l2_learning
```

Inputs to the modules are also possible via parameters:

POX module input

```
./pox.py log.level --DEBUG forwarding.l2_learning
```

To run the POX controller with the load balancer module, the following command should be executed from the pox directory. Do note that *–ip* is for the virtual server address and the *–servers* parameter is for the actual servers. There are no port options as this load balancer works in Network layer of the OSI model as shown in Figure 2.1.

POX load balancer example

```
./pox.py misc.ip_loadbalancer --ip=10.1.2.3 --servers=10.0.0.1,10.0.0.2
```

3.4.2 Floodlight SDN controller

Floodlight has been chosen as the second SDN controller as it also comes with a reactive load balancing algorithm. It is available at GitHub for download and requires JDK, Ant, Python and Eclipse that can be installed by the steps outlined in listings 3.2

Listing 3.2: Install and run Floodlight

```
1  $ sudo apt-get install build-essential default-jdk ant python-dev
       ↪ eclipse
2  $ git clone git://github.com/floodlight/floodlight.git
3  $ cd floodlight
4  $ git checkout stable
5  $ ant;
6  $ sudo mkdir /var/lib/floodlight
7  $ sudo chmod 777 /var/lib/floodlight
8  # Running the controller is done by with the following command:
9  $ java -jar target/floodlight.jar
10 # GUI is at: http://localhost:8080/ui/index.html
```

Default port for Floodlight is port *6653*, so running a minimal Mininet topology is done with the command bellow:

Mininet minimal for Floodlight

```
sudo mn --controller=remote,ip=127.0.0.1,port=6653
```

When the controller is running, configuring for load balancing is done using the REST API which is pushed to the controller using curl (HTTP) commands. The commands should be run via a script as shown in 3.3 (Wang, 2012)

Listing 3.3: Configure load balancing

```
1 #!/bin/sh
2 curl -X POST -d '{"id":"1","name":"vip1","protocol":"tcp","address":"
    ↪ 10.0.0.100","port":"80"}' http://localhost:8080/quantum/v1.0/
    ↪ vips/
3 curl -X POST -d '{"id":"1","name":"pool","protocol":"tcp","vip_id":"1"
    ↪ }' http://localhost:8080/quantum/v1.0/pools/
4 curl -X POST -d '{"id":"2","address":"10.0.0.3","port":"80","pool_id":
    ↪ "1"}' http://localhost:8080/quantum/v1.0/members/
5 curl -X POST -d '{"id":"3","address":"10.0.0.4","port":"80","pool_id":
    ↪ "1"}' http://localhost:8080/quantum/v1.0/members/
```

This script 3.3, will setup a round-robin load balancing algorithm based on connections for the virtual IP *10.0.0.100* at *port 80* with two servers in the back end pool. As a notation for the default settings for Floodlight, it loads many modules that can be seen in Figure 2.9. Reducing the amount of loaded modules, by removing the unused ones may improve the performance of this controller.

3.5 Experiments methodology

This sections breaks down the problem statement into tasks that needs to be completed and in the order they should happen in order to answer the problems statements.

1. Build and configure the hardware environment, which is already covered in the approach section.
2. Benchmark the hardware environment for baseline results
3. Determine link speed (bandwidth) parameter for virtual environment from hardware tests.
4. Build and configure the virtual environment
5. Benchmark reactive SDN solutions (POX vs Floodlight) using an automated Mininet high-level CLI approach.
6. Develop a proactive solution with focus on performance

3.6 Experiments Evaluation

The results gathered should be done so after an intent of what parameters we are searching for. This section covers these results.

3.6.1 Evaluating the performance of the solutions

Evaluation of performance is done in a client/server relationship. It is therefore necessary to run both client and server software. However,

there are not many complete solutions in the benchmarking world of client/server relationships. The most widespread ones are over the HTTP protocol, like the benchmarking tools httperf and ab.

Due to the environments not being fully comparable the numbers gathered cannot be directly compared. This is due to the fact that the system resources differ between the environments, meaning that doing baseline versus improvement of both systems in is a more sane methodology.

The tests are needed to determine the improvement for the load balancing scheme in comparison to not running one for both environments. This means that we must do performance test of both systems running HTTP-servers. To find out how does not using load-balance compare to using load-balance. I.E for SDN test using only load balancing algorithm and then forward algorithm.

For the httperf tests, what defines its rate limit; e.g when is a setup performing at max capacity, will be when the requested rate is no longer the same as the response rate. When defining parameters for httperf, it is intended to set the number of connections to a number that matches around 60 seconds for the test to complete. This is to ensure that the tests are more reliable than using short burst tests at only a few seconds. The actual performance parameters we are looking for are summarized in this list:

- Baseline results: How does Physical, POX and Floodlight handle normal switch behavior? Main parameter is: *Requests per Second*.
  - For virtual only: How does the Open vSwitch handle switching?
- Load Balancing results: How does the three setups handle load balancing?
- Is there an impact on the performance if there is small or large files being transferred?

3.6.2 Evaluating the scalability of the solutions

The performance of the systems is only part of what we aim to discover during our research, as another important part is the scalability of the solutions. To fully answer if the solutions are feasible, how they scale with increased resources is necessary. Like for instance that there is not any hard limitations associated with the solution. A hard limit is where the system fails to increase its performance even though it has increased resources. For example in a scenario where the application fails at a request rate of 50, it does then have some hard limit in it's programing, and therefore increasing the resources would not increase the rate above this number.

- For the virtual environment these are the questions we want to answer for the scalability of SDN:
  - How is the system resources used in addition to the throughput number of HTTP connections? Here we want to answer what is stealing the most CPU of the virtual environment? As it may be server software, benchmark software, Open vSwitch or the controller application. Finding the culprit could answer how the virtual environment would scale in a real environment.

3.7 Experiments

This section covers the exact test that are going to be conducted, that are in direct correlation with answering the problem statement. Other tests that are used to define network parameters are not listed here.

1. Small HTTP page test for all setups: Physical, POX(proactive/reactive), Floodlight(proactive/reactive)
   (a) httperf and ab test of baseline (1 server).
   (b) httperf and ab test of load balancing (2 servers).
   (c) When necessary, e.g results from test 1-2 does not show a result: load balancing (3 server).
2. Large load e.g big file transfer test for all setups: Physical, POX(proactive/reactive), Floodlight(proactive/reactive)
   (a) httperf and ab test of baseline (1server)
   (b) httperf test of load balancing (2 servers)
   (c) When necessary, e.g results from test 1-2 does not show a result: load balancing (3 server).

Part III

Conclusion

Chapter 4

Results and Analysis

In this chapter the results of the approach is presented. It contains the data collected from performance tests measured on both environments, as well as the analysis.

4.1 Results of initial link performance tests

As shown in Figure 3.2 the setup have two networks, one routed through a normal switch and one through the BIG-IP. This test is primarily about finding out what the actual link speed and latency of the physical links are, in order to set equivalent parameters in Mininet. However, as the BIG-IP lacks specialized hardware it is going to affect the performance. How the performance is affected is only discovered by testing a normal switch in addition to the BIG-IP unit. That is why routing through a switch will also be tested as a comparison.

4.1.0.1 Latency

Latency, measured in *ms* have been tested using the *ping* command with *-c* parameter *100* in Linux and a average score have been calculated and plotted into Figure 4.1 for both routes.

4.1.0.2 Network throughput

Network throughput, measured in *Mbit/s* have been tested using the iperf command. However, because of the network latency delay in the BIG-IP it needs more concurrent connections than what one iperf test generates to reach it's maximum. This means that multiple iperf tests must be issued at the same time to find the limit. By running those multiple iperf tests we found out that the limit for one link is around *700 Mbit/s*, but the BIG-IP can route more than this simultaneously, up to $\approx$ 1200 *Mbit/s*. But when all

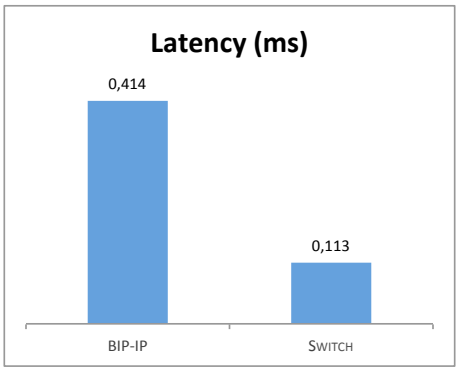


Figure 4.1: average latency comparison

four clients are pushing the four links it seems that the iperf test is affected by the latency, so that it does not show the true throughput.

For a normal gigabit switch the results from iperf shows that a link is not as fasts as it theoretical limit of 1 Gbps (1000 Mbit/s), but close at *"937 Mbit/s"* for both the UDP and TCP tests. This means that a physical link does *93,7* percent of it's theoretical limit on a normal switch. However, you should note that the 1GB link is pushing one billion bps, but that data transfers are based on some limiting factors such as frame size, which results in a lower *data* throughput.

4.1.1 Mininet network configuration of links

Because the BIG-IP is a Application Delivery Controller (ADC), without hardware acceleration it can not be directly compared against a switch in switching performance (Jesse, 2015). Because the 1600 don't have a Packet Velocity Acceleration (ePVA) chip, all traffic is processed in software, which a L2/3 switch does not. This means that under ideal conditions it will delay each packet in both directions by $\approx 20 - 29$ *ms* latency (Jesse, 2015). But for throughput you could overcome latency with concurrency, by increasing the number of traffic streams and unless the CPU reaches it limit, full throughput could be achieved (Jesse, 2015). This was however, not backed by our iperf results, as the highest throughput achieved was $\approx$ 660 *Mbit/s* on TCP and $\approx$ 740 *Mbit/s* for UDP. Do note that this difference in speed was not noticeable in the normal switch test scenario between UDP and TCP. A range from 1 to 10 simultaneously instances of iperf was tested but the total speed were not increased above these numbers, only divided among all the clients when over 4 simultaneously connections. The best-case scenario *740 Mbit/s* will be used as the link parameter in Mininet.

With regards to latency, the Mininet FAQ states that the link delay speed should be sat four times lower than what the intended delay is, as a packet travels through four links for a client to server test. This means that the ping delay of *0.414 ms* should be $0.414/4 \approx 0,1$ *ms*.

4.1.1.1 Parameters tested versus the real parameters

To check that the suggested theoretical parameters from the tests match up to the correct parameters in Mininet, they where tested and adjusted accordingly. The test results shows that in order to achieve $\approx$ 740 *Mbit/s* in Mininet, the link parameter *bw* should be sat to *820*, and for the delay to match it must be *0,07 ms*.

4.1.2 Mininet CPU limits

In a virtual environment like Mininet the CPU is by default shared equally between the hosts, switches and the controller in the network after demand. This is however not valid for a performance test where the amount of servers increase. The reason behind this is that they share the same amount of resources. So by increasing the amount of servers, we are only increasing the complexity of the system without increasing the capacity, which is the complete opposite of what load balancing aims to achieve. Because this was shown in early performance tests, a CPU limit has therefore been enforced in the run-time scripts as shown in the next section. The goal for the CPU limits is around 10 % CPU utilization for each host, this means that when increasing the amount of servers the resource pool is actually increased.

4.1.3 Mininet configuration file for run-time parameters

To automate the setup of the virtual environment, a Mininet high-level python CLI file has been developed. This script when run, initializes the virtual environment with the correct link parameters, the correct controller modules, the hosts and the server applications if wanted. This file is shown in Appendix A.1, and is started with the command *sudo python poxfile.py* for the POX version.

The other version is for the Floodlight controller, as shown in Appendix A.2. It is basically the same as the POX version expect that it is tailored for another controller, namely the Floodlight controller. It is started with the same command, with only the name of the file being different: *sudo python floodfile.py*.

These scripts may also take other commands as parameters, for instance like starting a httperf of ab benchmark of the environment. An example of doing so is shown in the POX Appendix A.1.

4.2 Performance test results for the physical environment

This section covers the results gathered from the performance tests executed for the different scenarios as outlined in the approach. The results gathered from the physical environment is presented in this section. There are five test scenarios:

1. BIG-IP acting as switch from host to one server.
2. BIG-IP acting as a load balancer for one server.
3. BIG-IP acting as a load balancer for two servers.
4. BIG-IP acting as a load balancer for three servers.
5. BIG-IP saturation point.

4.2.1 httperf tests

For the httperf test, it was unable to find the saturation point due to the fact that it exits with error code *"*** buffer overflow detected ***: httperf terminated"* when running the following command, which generates less traffic than the system maximum:

httperf command at overflow parameters

```
httperf --server 10.20.0.100 --uri '/index.html' --num-conn 13680 --num-call 100 --rate 228 --timeout 5
```

Do note that this was for a custom version of httperf, as the Ubuntu packaged version fails with different error related to maximum number of open file descriptors, at even lower loads. The error message the Ubuntu version outputs, is attached below:

httperf error in version from Ubuntu repository (0.9.0-2)

```
httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
```

This means that no valid httperf test could be gathered from the physical environment, as the software has errors in it.

4.2.2 ab tests

This section covers the gathered results from benchmarking the physical environment with the benchmarking tool ab. The BIG-IP can run different forwarding options for load balancing, but the standard option have been selected for the tests.

4.2.2.1 Few bytes test

This first physical test is for HTTP-servers only serving a small HTML page at 94 bytes. The results from this test is shown in Table 4.1 and from it we see that for using the direct IP to connect, bypassing the load balancing algorithm for one server is slightly faster than using the load balancing algorithm in the F5. The most noticeable difference however, is in terms of CPU usage as shown in Table 4.2, as load balancing uses 44% CPU in comparison to only 16% when only acting as a switch.

For two and even three servers in the back end, it is clear that the initiator is at it's max capacity. Because increasing the server pool so that they only operate at 77% capacity (Table 4.2) does not induce more load on the BIG-IP or provide any significant increase in performance. This is shown by multiple parameters, as ≈ 500 more requests per second is not a significant improvement. The same holds true for 3 servers being 1 second faster and outputting ≈ 200 more Kbytes/s. Finally this is backed up by the fact that the amount of concurrent requests is not increased as shown in the Table 4.1.

Table 4.1: BIG-IP, results for few bytes (94)

	RPS	*Duration*	*Kbytes/s*	*ms*	*-c*	*-n*
(1 server) Direct	**14675**	**61**	**5188**	**0.068**	**670**	**900000**
(1 server) Via f5	**14646**	**61,5**	**5178**	**0.068**	**650**	**900000**
Load balance (2 servers)	**20302**	**44**	**7177**	**0.049**	**1300**	**900000**
Load balance (3 servers)	**20805**	**43**	**7355**	**0.048**	**1300**	**900000**

Table 4.2: BIG-IP, results for few bytes (94) CPU load

	BIG-IP CPU load	*HTTP Servers CPU load:*
(1 server) Direct	Core1=16%, Core2=16%	100 %
(1 server) Via f5	Core=1=44%, Core2=44%	100 %
Load balance (2 servers)	Core=1=56%, Core2=56%	92 %
Load balance (3 servers)	Core=1=56%, Core2=56%	77 %

Because the initiator fails before the BIG-IP its saturation point is unknown. As the only proven thing is that the BIG-IP works, as a load balancer the next thing is the actual saturation of the BIG-IP. But because the previously tests has shown that the initiator is not capable alone of pressuring the BIG-IP to the maximum, the next test was done using three servers and running different -c parameters until we had enough to draw a line showing where the BIG-IP most likely would fail.

In this BIG-IP saturation tests the -c parameter was sat to one, and then increased until no higher performance could be met. The recorded parameters were then: *Concurrent Requests*, *Requests per Second* and the *CPU usage* of the BIG-IP. Because the memory usage stays below $\approx 100MB$ for all

tests this has not been taken into consideration, due to the fact that available memory is *1400 MB* for each core. System usage of back end servers are also not considered as it stays below $\approx$ 77% for all tests. As a final precaution network bandwidth is not an issue either, as the test at maximum uses approximately *26 MB/s*. The data recorded is for 1-256 increments of 2x, as shown in Figure 4.2. It increases until *-c 64* before at *-c128* it stays the same and *-c256* degrades the performance. From the calculated line in the figure, the BIG-IP would be able to load-balance approximately *40000* requests before being at maximum capacity.

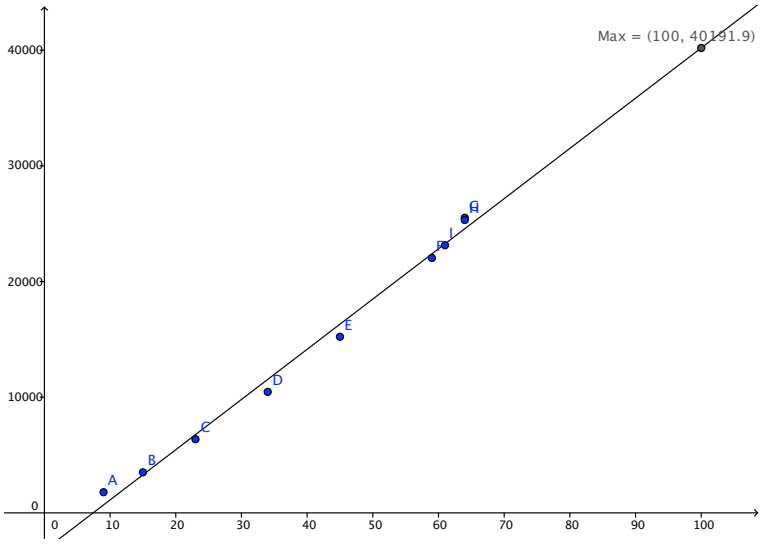


Figure 4.2: BIG-IP calculated saturation, after formula: $434.15x - 3223.52$. Showing RPS and CPU usage.

What is shown from these tests is that the physical load-balancer works as intended. It efficiently increases the RPS the system can handle and it doubles the concurrent requests possible from *650* to *1300* as shown in Table 4.1. Given enough backend servers, and a fast enough initiator it should be able to increase the capacity in a linear line up to its maximum utilization of the CPU.

4.2.2.2 1 MiB test

This test was conducted in the same way as for the few bytes test; by running ab with *-concurrency* parameter from 1 and up to 32 on a 1 MiB file served by the HTTP backend servers. Measuring the 1 MiB test for the physical environment reviled that for this test the Ethernet bandwidth was the limiting factor. This is shown by multiple parameters in Table 4.3, for instance that the Mb/s does not increase from *90 MB/s*, neither does the BIG-IP CPU usage and as stated in approach when the RPS is going down the system is saturated. However, in this case the link is the saturated part, which means that this test won't explain anything about load balancing, as there is only one outgoing link to the initiator.

Table 4.3: BIG-IP, results for 1 MiB

- c	RPS	BIG-IP CPU	Mb/s (max)
1	63	16	70
2	82	26	88
4	70	25	90
8	66	25	90
16	67	26	90
32	68	25	89

4.3 Performance test results for the virtual environment

This section covers the results gathered in the virtual environment with four CPU cores. The parameters gathered here, are used to determine what impact that the different controllers have to the viability of SDN-based load balancing. To produce these numbers the tests were conducted until a stable saturated result was achieved. This means at least three tests and the mean value was taken out. In many cases the results has been rounded to the closest number due to the inconsistency of using decimals in these tests. In addition to the fact that the variance is not crucial to the collected data as results compared against each other are not conclusive unless they show a clear pattern. In shorter terms, if one solution is for example only 1 second faster than another it is not conclusive or part of the solution as we are looking for clear tendencies with running the different controllers.

4.3.1 Mininet performance issue

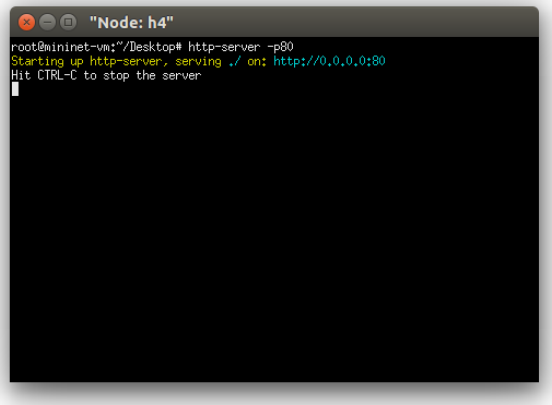


Figure 4.3: XTerm CLI starting HTTP server

Initially did the script created in Section 4.1.3 boot up the services that

needed to run in the environment for it to be complete. However, what was discovered was that when the HTTP server was started in this fashion the performance did equal the performance of manually starting the server via the XTerm CLI window. For our normal environment two tests are listed below, one for manually starting the HTTP server (4.2) and one for letting the script automatically starting it in the background (4.1). The test listed is with httperf and POX, but both controllers and benchmarking tools produced the same results.

Listing 4.1: Automatic started

```
1 Total: connections 2520 requests 830 replies 555 test-duration 133.878
    ↪ s
2 Request rate: 6.2 req/s (161.3 ms/req)
3 Errors: total 1965 lient-timo 1965 socket-timo 0 connrefused 0
    ↪ connreset 0
```

Listing 4.2: Manually started

```
1 Total: connections 2520 requests 2520 replies 2520 test-duration
    ↪ 126.051s
2 Request rate: 20.0 req/s (50.0 ms/req)
3 Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
```

For the automatic test, as shown in Listing 4.1 the rate is well below the expected performance for this environment at a rate of *20 requests per second*, however of the *2520* requests sent *1965* of them did result in client timeout error in httperf. This is not acceptable as the expected result is *0* client timeout errors as shown in the result from the manually started test in Listing 4.2. This means that for all performance tests the HTTP-servers was started manually as shown in Figure 4.3 to avoid this limitation.

4.3.1.1 Solution

The solution to this lies within the console output buffer, the background terminals windows have. Every server or test that runs in the background must either output no information at all or redirect all its output to somewhere else than the console, for instance like a file. This means that the console server like python's SimpleHTTPServer cannot be used, as it has no silent operation mode.

4.3.2 POX Controller

This section covers the results gathered form benchmarking the POX SDN-based controller as described in the approach Section 3.7.

4.3.2.1 Results for tests with a small HTTP page (few bytes)

The first test was completed with ApacheBench (ab) and is shown in Table 4.4. This table is for the results of running no load balance (1 server)

and then running the load balancing component for 2 and 3 servers. The final line is for running both components, meaning that the forwarding component and the load balancing component are both loaded at the same time. It is included to show a real use case scenario, as a SDN controller should as a minimum do normal packet forwarding in addition to load balancing. The ab parameters shown in the table are explained in the background section at Section 2.11.7.

Table 4.4: POX Controller, results for few bytes (215) AB

	RPS	*Duration*	*Kbytes/s*	*ms*	-c	-n
1 server	**44**	**64**	**22**	**22**	**700**	**2820**
2 servers	**102**	**58**	**50**	**9**	**1400**	**5960**
3 servers	**142**	**46**	**71**	**7**	**2100**	**6560**
both	**0,33**	**61**	**0,16**	**3066**	**1**	**20**

The performance results from the ab tests on the POX controller from Table 4.4, shows that when only one component at the time is loaded, the POX controller performs like a BIG-IP load balancer. This is shown because the concurrency parameter doubles from one to two servers and the response time is reduced in half. For the other test with both components loaded, where the controller load balances and forwards packets, the results are low in comparison to the tests with only one component loaded.

The second series of tests were completed using the tool httperf and the results are shown in Table 4.5. The test is the same as for ab, except that the software is different, resulting in other output parameters. Besides that, this table follows the same structure as for the ab test. For an explanation of the columns please consult the httperf Section 2.11.6 of the background.

Table 4.5: POX Controller, results for few bytes (215) httperf

	Num-conns	*Request rate asked: (req/s)*	*Completed rate*	*Duration*
1 Server	**2 820**	**47**	**46,6**	**60,54s**
2 Servers	**4860**	**81**	**80,3**	**60,45s**
3 Servers	**6000**	**100**	**85**	**70s**
both	**30**	**2**	**1,6**	**14,5s**

For the httperf test, system usage was also recorded. How much CPU was used is shown in Table 4.6. Do note that the results were very fluctuating so it is therefore shown as the range they varied within during the course of the tests. For all tests the switch CPU usage is very consistent with the request rate being processed, as it close to doubles as the rate increases from 1 to 2 servers. It also stays consistent as the rate is only marginally increased from 2 to 3 servers, by only using a bit more CPU. However, for the controller the load balancing module drastically increases its CPU usage. And because the controller have peaks that uses almost all the

available CPU, the rate does not increase as it did from 1 to 2 servers. The limitation here is therefore the available CPU the controller have.

Table 4.6: POX Controller, results(few bytes) httperf CPU percentage

	Switch CPU usage %	Controller CPU usage %
1 server	8-25%	7-15%
2 servers	40-50%	40-55%
3 servers	45-50%	40-90%
both modules	1-2%	1-3%

4.3.2.2 Results for tests with a large HTTP page 1 MiB

This section covers the results gathered from the tests where the file transferred is much lager than the first test, which was only a few byte. In this test the file transferred is *1048576* bytes ($\approx$ $1MB$), that is *4877* times lager than 215 bytes for the first test. This is meant to simulate a scenario for large file transfers or heavy load on the back-end for each client request. In these results as shown in Table 4.7, there are no results for both modules loaded as the test with both modules loaded crashes the controller application and the following error message is outputted for the switch:

Error message from OVS

```
|ERR|s1<->tcp:127.0.0.1:6633: no response to inactivity probe after 5 seconds, disconnecting
```

This error message is related to the OVS switch not being able to talk to the SDN controller. This is because the controller has stopped responding. In some rare cases the OVS proses *ovs-vswitchd* crashes as well because of this error. Do note that for this test, only the benchmarking tool httperf was used as the *few bytes test* has shown that both applications show the same trends.

Table 4.7: POX Controller, results for 1 MiB httperf

	Num-conns	Request rate asked: (req/s)	Completed rate	Duration
1 Server	**120**	**2**	**2**	**59,9**
2 Servers	**240**	**4**	**4**	**60,1**
3 Servers	**360**	**6**	**6**	**60,2**
Both	**x**	**x**	**x**	**x**

For the $\approx$ $1MB$ test one server was able to serve *2* requests each second as shown in Table 4.7. When a higher rate of *3* was tested, it resulted in *2,8 req/s* but with almost all requests timing out. The same applies to the 2 and 3 server tests, so the rates shown in Table 4.7 are the limits for this scenario without errors as described in the approach.

Table 4.8: POX Controller, results for 1 MiB CPU usage httperf

	Switch CPU usage %	Controller CPU usage %
1 server	3%	2,5%
2 servers	4%	3,5%
3 servers	5%	4,5%

As shown in Table 4.8, the POX controller uses a low amount of CPU when the requests are much more intensive on the servers. This is because it needs to install fewer flow rules than for small HTTP requests.

4.3.2.3 Results for tests with a large HTTP page 2 MiB

Because of the CPU limitations and the transfer time that follows the 2 MiB test, it is the largest test supported in the virtual environment. Because performing tests on 3 MiB or more only results in one server not being capable to transferring the files without being considered timeouts.

Table 4.9: POX Controller, results for 2 MiB httperf

	Num-conns	Request rate asked: (req/s)	Completed rate	Duration
1 Server	**60**	**1**	**1**	**59,8**
2 Servers	**120**	**2**	**2**	**60**
3 Servers	**180**	**3**	**3**	**60,3**

The results from the $\approx 2MB$ test, shows that the load balancing algorithm is working. It is increasing the possible rate from *1* for one server, with *1* for each server that is added. From the raw output it is able to achieve *1.8,2.8,3.8* as the completed rates when asking for 2,3,4 on the 1-3 server scenarios, but that results in more than a 10% error rate for each test, which is not feasible. So the maximum feasible rate achieved is presented in Table 4.9. In this table it shows some duration differences, but it is negligible as more results gathered only produces a duration result average closer to 60 seconds.

Table 4.10: POX Controller, results for 2 MiB CPU usage httperf

	Switch CPU usage %	Controller CPU usage %
1 server	2%	1,5%
2 servers	3%	2,5%
3 servers	4%	3,5%

4.3.3 Floodlight Controller

4.3.3.1 Results for tests with a small HTTP page (few bytes)

The same tests where done for the Floodlight controller as they were with POX. In Table 4.11 the tool ab was used to benchmark 1 and 2 servers. Do note that because Floodlight runs forwarding by default, there is no additional test with both modules loaded. This is because load balancing is applied on top of the normal forwarding using a curl script. Running the tests this way is also the most realistic scenario for using the SDN controller. The ab results show that running the load balancing module on top of the forwarding module reduces the service performance from *40 RPS* to *20 RPS*. It also reduces the possible concurrent requests from *500* to *1*.

Table 4.11: Floodlight Controller, results for few bytes (215) ab

	RPS	*Duration*	*Kbytes/s*	*ms*	*-c*	*-n*
1 Server	**40**	**70**	**20**	**25**	**500**	**2820**
2 Servers	**20**	**25**	**9,8**	**50**	**1**	**500**
3 Servers	**20,5**	**24**	**10**	**48,8**	**1**	**500**

Following the same structure, benchmarks were also completed using the httperf tool on Floodlight and is shown in Table 4.12. For this test 3 servers in back end was also tested, but as the table shows increasing the number of servers did not increase the number of RPS completed. These httperf results shows the same pattern as ab did, which is; that when applying load balancing the performance goes down from normal forwarding. In this case it goes down from *50 RPS* to *9 RPS* and stays at that rate even as another server is added to the pool.

Table 4.12: Floodlight Controller, results for few bytes (215) httperf

	Num-conns	*Request rate asked: (req/s)*	*Completed rate*	*Duration*
1 Server	**3060**	**51**	**50**	**62**
2 Servers	**540**	**9**	**9**	**60**
3 Servers	**540**	**9**	**9**	**60**

As it is a part of the performance of the controller, the system CPU usage was recorded for the httperf tests of Floodlight as well, this is shown in Table 4.13. Do note that the results were fluctuating so it is therefore shown as the range they varied within during the course of the tests.

Floodlight does not stress the CPU according to Table 4.13. It does however have some high peaks in CPU usage up to 20 % for the load balancing algorithm.

Table 4.13: Floodlight Controller, results(few bytes) httperf CPU percentage

	Switch CPU usage %	Controller CPU usage %
1 server	30%	10-14%
2 servers	2-4%	5-6% (normal), 10-20% (peak)
3 servers	2-4%	5-6% (normal), 10-20% (peak)

4.3.3.2 Results for tests with a large HTTP page 1MiB

For this test the Floodlight controller was booted up in normal mode which includes a forwarding module for the test for 1 server. For the 2 servers test the same curl script as for the *few bytes test* were applied to enable load balancing over 2 and 3 servers. The result of this experiment is shown in Table 4.14. Do note that for this test only the benchmarking tool httperf was issued as the *few bytes test* has shown that both applications show the same trends.

Table 4.14: Floodlight Controller, results for 1MiB httperf

	Num-conns	Request rate asked: (req/s)	Completed rate	Duration
1 Server	**120**	**2**	**2**	**59,9**
2 Servers	**120**	**2**	**2**	**59,8**
3 Servers	**120**	**2**	**2**	**59,8**

Some surprising results occurred for the $\approx 1MB$ test as it is shown in Table 4.14. Because starting the load balancing algorithm does not increase the number of large requests possible. The difference in duration is negligible, meaning that for the 1 MiB test all tests with Floodlight performed the same. The most noticeable difference for these tests is that the CPU usage as shown in Table 4.15 has a higher peak when using the load balancing algorithm.

Table 4.15: Floodlight Controller, results for 1MiB CPU usage httperf

	Switch CPU usage %	Controller CPU usage %
1 server	3%	2-4%
2 servers	3%	3,5% - Peak $\approx$ 10%
3 servers	3%	3,5% - Peak $\approx$ 10%

4.3.4 SDN-based Load Balancing Results

4.3.4.1 Small HTTP page

This section is a summarization about how the SDN controllers performed as load balancing unit for a few bytes. Starting with Figure 4.4, shows that

for POX changing the forwarding module to the load balancing module does in fact increase the performance of the system as a load balancer should. This is shown by one server doing *44* RPS and two servers doing better as the load is shared between them and that three servers doing approximately one server better in RPS than two servers. This is consistent with the results from the physical environment, and the general idea about load balancing. It is easier to spot in the transfer rate as it increases in clearer increments from one to three servers as shown in Table 4.4.

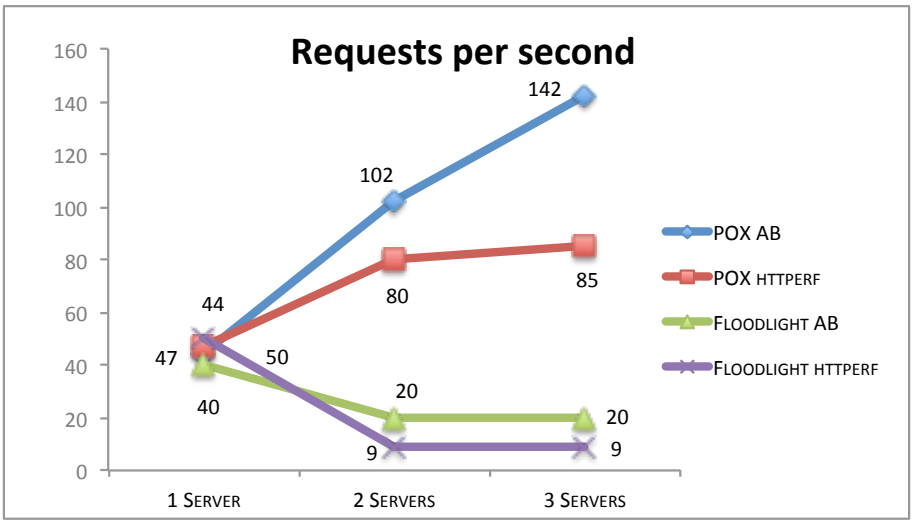


Figure 4.4: SDN as a Load Balancing algorithm results

For Floodlight the results in Figure 4.4, are worse than for POX due to the fact that POX could run one component at at a time. Because when enabling load balancing on the running Floodlight environment the performance of adding servers is actually degraded, which is the opposite of the desired effect. Adding computing resources should result in performance gain to be a viable solution. But Floodlight handles $\approx$ 50 RPS in the ab test for one server and then degrades the performance of the system when introducing load balancing down to about *20* RPS. The controller CPU usage from Table 4.13 shows that the CPU was not fully utilized during the test. This means that the slowdown in Floodligth is not in the CPU, but as a hard limit in the software code.

For Floodlight the benchmark tools are not presenting the exact same numbers, but they do show the same trend. This is only partially true for POX as the httperf test does not curve the same way as ab in Figure 4.4. The explanation for these results is that all virtual hosts have the same 10 % CPU allocation, but because httperf is a more CPU consuming application than ab. It is not capable to producing the same pressure as ab is. This is shown by process monitoring and httperf output detailing the usage split between user and system, which is *(user 9.5% system 0.5% total 10.0%)* for the 3 server test of httperf. The 10 percent is the limit the virtual hosts have on the CPU, and it was used by httperf in the ratio *9.5/0.5*.

4.3.4.2 Large/intensive HTTP requests

This section is a summarization about how the SDN controllers performed as load balancing unit for a lager load of $\approx 1MB$. What is clear from Figure 4.5 is that as the servers increase, so does the performance while using the POX controller. This is however not a strictly fair comparison as Floodlight must run both modules, but there is no way of disabling the forwarding module in Floodlight as it is a dependency for the load balancing module.

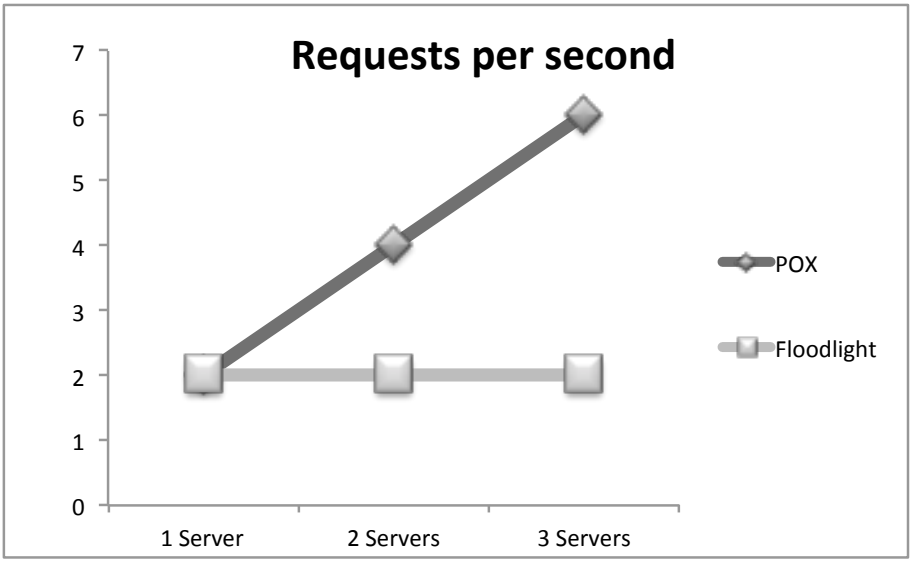


Figure 4.5: SDN as a Load Balancing algorithm results for large load

4.3.5 Proactive POX load balancing

As the previous results involved benchmarking of reactive solutions, this section covers the development of a proof-of-concept solution, for proactive POX load balancing for performance benchmarking purposes. As version lower than 1.5 of OpenFlow does not support port matching on TCP flags (e.g. SYN, FIN, and RST) the proactive solution can't account for the status for a TCP connection (OpenFlow, 2014). This is also the case for the reactive solutions that have been tested. This makes connection migration from one server to another harder, but it is not considered at this stage.

The solution uses the *arp_responder* and *forwarding.l2_learning* component from POX to provide what would be a realistic environment where normal forwarding and load balancing is enabled. The forwarding component comes bundled with the latest version of POX, but *arp_responder* has to be manually installed from source (J. McCauley, 2013) as it is not included in latest release of Mininet. This is done by coping the component into the *ext* folder in the pox directory. The component file created is shown in

Appendix A.3. It is compatible and used in conjunction with the Mininet start up script presented in the Mininet configuration section. It is used in the next section, which covers the proactive load balancing benchmark results. For the development of this script, tcpdump and Wireshark was use to debug the traffic flows, it was by using these tools we were able to determine that the order of the commands for the flow rule builder in POX matter. As during tests packets arrived at the correct machine but did not have their IP correctly rewritten. This did mean that the receiving server did not respond to the packets, but the packets were visible with tcpdump at their networking interface. This is noted in the script, to avoid any confusion for the users.

4.3.5.1 Results for 1 server

For this test 9 data points was captured and standard deviation, confidence interval at 90 percent using t.test and mean was calculated. The calculated data is shown in Table 4.16, and from it it is clear that the completed rate and duration is a reasonable accurate number as the standard deviation is very low. Our confidence interval of *60,93661-61,48561* would contain the average of all the estimates 90% of the time if we continued the experiment additional times in the exact same way.

Table 4.16: Proactive POX httperf test of 1 server at saturation point - Request rate requested: 42 (req/s), B.1

	Num-conns	Rate requested:	Completed rate	Duration
Standard deviation	2520	42	0,355555556	0,296296296
90% CI (t.test)	2520	42	40,98262-41,5507	60,93661-61,48561
Mean	2520	42	41,26667	61,21111

As with the other tests, the CPU usage was also recorded and for one server the switch used *5-7* percent CPU. The controller used only *0-2* percent CPU at operation as when the rules are installed it does not need to do any specific actions.

4.3.5.2 Results for 2 servers

For this test 8 data points was captured and standard deviation, confidence interval at 90 percent using t.test and mean was calculated. The calculated data is shown in Table 4.17

The CPU usage for two servers is at this point close to linear at about *11-13* % used by the switch. The controller does not use more CPU than it did for the 1 server test, this means that it stays at *0-2* percent.

Table 4.17: Proactive POX httperf test of 2 servers at saturation point - Request rate requested: 84 (42x2) (req/s), B.2

	Num-conns	Rate requested:	Completed rate	Duration
Standard deviation	5040(2520x2)	84 (42x2)	0,34375	0,080622222
90% CI interval,(t.test)	5040(2520x2)	84 (42x2)	82,99482-83,75518	60,23185-60,31815
Mean	5040(2520x2)	84 (42x2)	83,375	61,21111

4.3.5.3 Results for 1 and 2 servers (1 MiB)

The test with 1 MiB is presented in Table 4.18. Here the controller has a low CPU usage at about *0-2* percent. As for the switch it uses $\approx$ 2% for 1 server and $\approx$ 4% for 2 servers. This means that the switch CPU usage scales with the RPS completed.

Table 4.18: Proactive POX httperf test of servers at saturation point for 1 MiB

	Num-conns	Rate requested:	Completed rate	Duration
1 Server	120	2	2	60,03
2 Servers	240(120x2)	4(2x2)	4	59,98

4.3.5.4 Proactive solution

The proactive solution where rules are installed into the switch before a packet arrives gives a linear curve for this test scenario as shown in Figure 4.6. Because this solution does load balancing based on source IP for one virtual IP (VIP), there must be on client for each server. This also means as long as the CPU usage of the system is not at maximum, it is possible to continue to add new servers and clients to get more RPS total. One notation is that the CPU usage of the controller stays at 1% as it does not need to actively interact with the switches.

As for the test with bigger load, it behaves in the same fashion as for the small HTTP requests, because increasing servers does linearly increase the RPS possible.

4.4 Solutions comparison

As we know that the proactive solution scales linearly, the 3 server results for proactive can be calculated, and all httperf results can be inserted into a graph. This has been done in Figure 4.7 and it shows how the performance of the proactive and reactive solutions. What is clear is that none of these setups are without their drawbacks. For instance does the proactive solution provide best performance. However, it will require

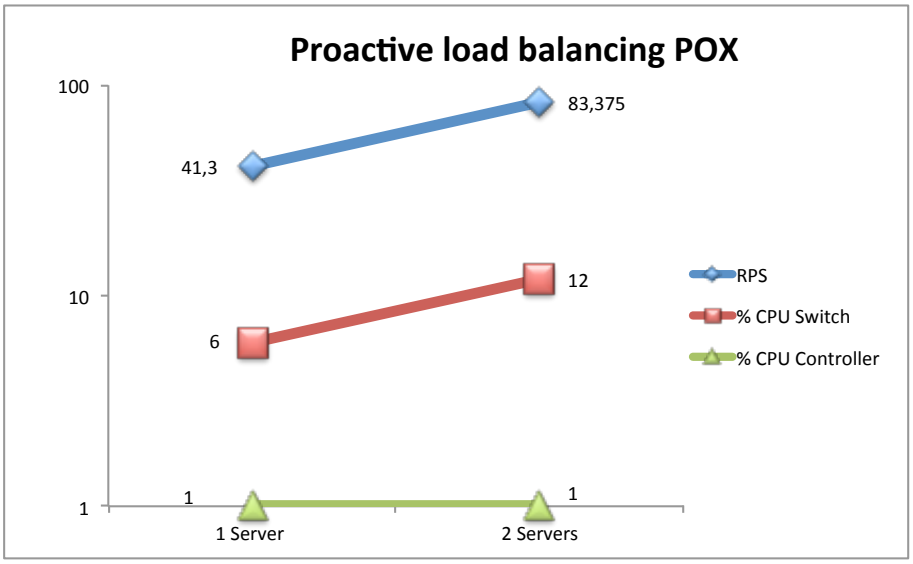


Figure 4.6: Proactive load balancing POX for few bytes in httperf. Figure in base10, shows relativity between CPU usage of switch and RPS achieved.

an algorithm to update the flow tables over time to ensure reliable load balancing. For instance in cases where a server goes down and the client IP range associated with that server would need updated flow tables for the traffic to be transferred to the a new server. This is to ensure that the clients are not sent to a server that is unavailable. The reactive POX solution offers reasonable performance until the controller is overloaded, but it does not provide load balancing and normal forwarding at the same time. While as the reactive Floodlight solution does provide both forwarding and load balancing, the performance throughput for a few bytes is very low compared to what the load balancing components can do alone.

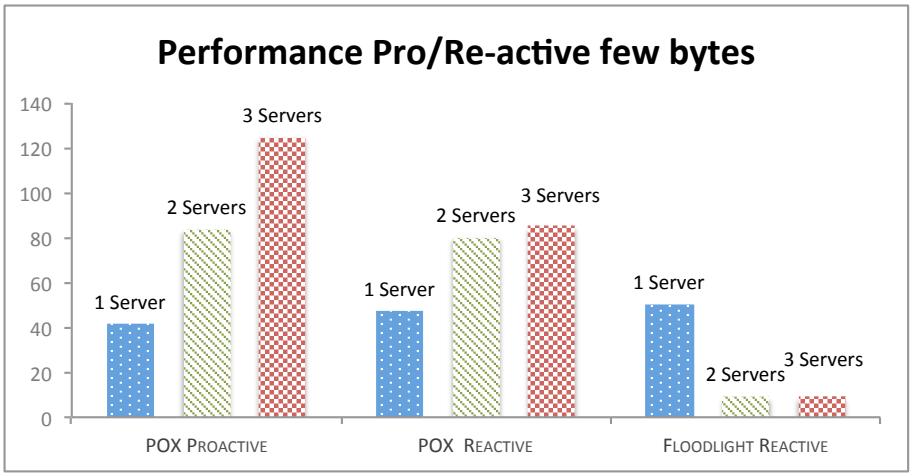


Figure 4.7: Proactive and reactive performance diagram for httperf tests of few bytes

As seen in Figure 4.8 the proactive solution is not faster than the reactive for large loads. This is shown as both solutions performed at RPS rate of 2 for each server added. It does however reduce the amount of CPU used by the POX controller from $\approx 3,5\%$ down to $\approx 1\%$.

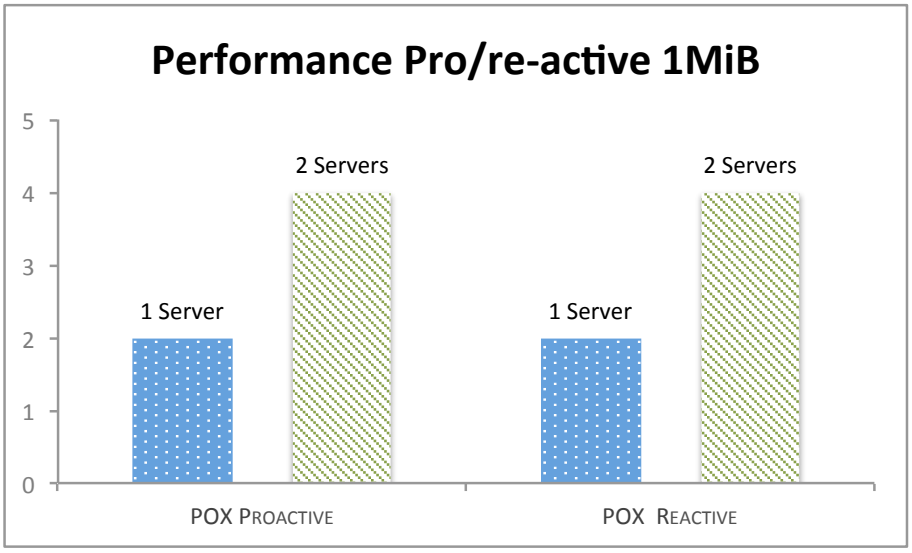


Figure 4.8: Proactive and reactive performance diagram for httperf tests of 1MiB

4.5 Solutions analysis

The first results gathered were used as a baseline to measure the results against each other. What was gathered from the first physical environment tests was that the introduction of load balancing should close to double the performance of the system. That should therefore be the metric to determine if it is feasible to move the load balancing from a dedicated unit into the SDN network.

During tests the virtual switch in Mininet, Open vSwitch have performed exemplary. Only during one test did the switch crash, where the fix was a system restart. It was never the slowest part of the environment, so it cannot be the source of any variation of the results. Because of differences in programing language and programing logic, two controllers should give a deeper understanding of if SDN is mature enough to replace a dedicated unit. And because they did performance similar when in the same conditions it is enough to draw a conclusion about the SDN load balancing status.

For the tests with a small HTTP page being load balanced, representing a service where there is a high number of requests, did both controllers perform in a similar fashion. The POX controller performs best with only the load balance component loaded as it doubles the system performance

from 1 to 2 servers as load balancer should. What is disappointing however is that when POX is started in a realistic scenario with both forwarding and load balancing it can not sustain a feasible RPS or kbytes/s throughput for both the httperf and ab test. What is the most surprising about this, is that it is not a system resource limitation as it only uses a few percent of the CPU, so the limit here lies within the programing itself. This means that it won't be a viable solution in a full-scale environment as adding more resources would not solve this limitation of the setup. The odds are further stacked against SDN load balancing as the Floodlight setup encounters the same limit with both components loaded as POX did. That unfortunately means that the slight performance advantage it has is negligible as that is just for the baseline results without load balancing.

The results for the scenario with large file transfers are where the controllers show their clear differences. Both components are loaded on both controllers for this scenario and in these results POX is actually functioning as a load balancer where as Floodlight is not. Because POX has a linear expansion curve as shown in Figure 4.5, however for Floodlight in this figure the line is flat. The line is flat because Floodlight does not manage to increase the RPS for the large file transfers scenario.

This means that, if choosing between POX and Floodlight that POX is the obvious choice for large file transfer load balancing. It however also proves that none of the tested SDN controllers should be used as a replacement for a dedicated hardware load balancer for small file transfers/requests. With that in mind it, from Figure 4.7 it seems feasible to run a proactive load balancer using the POX controller. Because doing so, the only real limitation is in the switches themselves and not in the controller. It would however mean that for it to be used in a production environment, that it were more advanced so it could handle a larger variation of incoming clients and automatically distributing the source IP ranges over the available back-end servers based on availability.

If the traffic that needs to be load balanced is for large file transfers or computing heavy requests, so that the amount of concurrent requests are lower than for instance a typical web page with a high number of users it seems reasonable to run SDN load balancing but only as long as there is no forwarding needed. If the network needs a forwarding controller as well, as a typical production network does, using SDN load balancing for large requests is not feasible. As in terms of scalability of the solutions, does the controllers use the most CPU. This is because the clients and servers are limited, but the controller application could use as much CPU as the virtual environment had available. This means that for a full-scale solution if the controller needs to work in a reactive mode, the network throughput is solely dependent on the speed of the controller.

Chapter 5

Discussion and Future Work

5.1 Discussion and Conclusion

The HTTP tests used for benchmarking have produced valid and intended data. The results are valid because they show clear patterns of the tested solutions and from the results themselves there were no surprises. Although, in what the results told when analysed was surprising with regards to the controllers using multiple components. Some comparable data is hard to find, as the BIG-IP does not have a HTTP load balancing rate listed in its hardware specification sheet. This is also true for the virtual environment, as the load balancing algorithms have not been tested in this type of setup. To reproduce the same results for another researcher would be trivial for the virtual environment, as it only uses free to use tools. However, for the physical environment a BIG-IP LTM device would be an expensive hardware to acquire for the same test.

5.1.1 Network topology

All the papers explored in the related work section, that had some success with SDN-based balancing tested unstructured network topologies, where there is no clear source of client requests. There were, however no papers that tested a scenario as if SDN could replace a dedicated hardware load balancer in a structured or random network. However, this thesis only looks into a structured network, because that is the main configuration for a dedicated load balancing unit.

5.1.2 Feasibility cases for SDN

This thesis provides predictions for the feasibility of replacing a LTM unit as the BIG-IP with an SDN-based solution. There are three cases that it is possible to make predictions about:

1. Small requests

    (a) Requests that have short processing time or small packets being sent with a high amount of users.

2. Big requests

    (a) Requests that uses a lot of bandwidth, like a large file transfer.

3. CPU intensive requests

    (a) Requests that are computationally heavy, e.g that the back-end servers use a long time for processing so that the amount of users are lower than for small packets with a high amount of users.

The two last cases, 2 and 3 are possible to merge as they are similar in the way that the controller only needs installed a few flow rules. What was shown in the analysis Section 4.5 with regards to these cases is that SDN-based load balancing is probably not in the near future going to replace a dedicated unit for *case 1*. Because both tested solutions did not handle a high RPS satisfactorily, and would clearly have some limitations with respect to system resources. This limitation is based on the fact that the dedicated unit has a higher CPU to RPS ratio even just for forwarding, which was what the controllers supported for best performance.

With regards to case 2 and 3 they are much more promising as Section 4.5 outlined. This is because the CPU usage is much lower, and when only running a load balancing algorithm their speed is linearly increasing as long as there is available CPU. However, though the SDN controllers have a hard limit, the programing limitations are their biggest obstacle for becoming feasible. This hard limit in software was a big surprise as some incompatibility or high CPU usage was to be expected, but not such as breakdown as the controllers did with both modules loaded.

5.1.3 Load Balancing algorithms

For the most promising controller POX, the load balancing algorithm supplied picks servers in a random fashion. This cause more unpredictable results as it may easier overload one of the backend servers. This resulted in that a lot of the tests had to be remeasured to ensure that the result was stable. In contrast to the BIG-IP that does not have a random algorithm, it did prove to give more stable results as the algorithm *round robin* divides the load better. The other controller Floodlight, also provides load balancing after the *round robin* principle. However, as the controller struggles with a software limitation it does not impact the measured results in a positive way.

5.1.4 Mininet bug, implications on recorded data

A high level Mininet CLI script was developed to automate the benchmarking process. Unfortunately, an undocumented python/bash bug stop the servers after a given amount of console output. However, when servers are started with *h3.cmd('http-server -p 80 &')* instead of manually typed into the XTerm console the performance suffers. Starting the servers like that resulted in them stopping because of the output buffer being filled up. That is why all servers were initiated manually by typing *http-server -p 80* into the XTerm CLI. This is however a much more tedious and slower process as the environment is reset between each test for a clean test each time. The biggest impact of this bug for this thesis is that originally it was intended to use the Mininet high-level API function for this automated sequence.

1: Boot network. 2: Start services. 3: Do performance test. 4: Log data.

However as the solution to this limitation were discovered three days before the delivery date by consulting the Mininet development team, the script was not used to conduct the tests. But most of the script is already developed and included in this thesis, some reconfiguration would however be needed for complete automation as it was not completed because of this error. If the functionality had worked, collecting more datapoints (30+) for all tests would have been feasible. It does not however have a big impact in the value of the data as all collected data show clear tendencies. As for further tests, the variance in data is dependent on the host operating system. However, by using the provided Mininet scripts all data recorded on other systems will show the same graphs and percentage differences, with the baseline results varies with the maximum processing power of the host operating system. Using this script is strictly recommended, as it was a mistake not to consult the development team earlier to overcome this limitation.

5.1.5 MiB instead of MB

When the 1 and 2 MiB test files where created, it was the intention to create 1 MB files for the simple denomination. However the inputted multiplier was 1024 instead of 1000 for the *dd* command, resulting in 1 and 2 Mebibyte (MiB) size files. This error was not uncovered until multiple experiments had been conducted, and was therefore not corrected. The only implication of this however, is the denominator for the tests as the byte difference does not make any impact on the test as both controllers were tested using the same MiB files.

5.1.6 Problem statement

The main idea of the problem statement of this thesis was to find out if SDN-based load balancing could in some way replace a dedicated LTM unit. For this it also needed to configure an LTM device to have something to compare the SDN solutions to. This did however require time which if not needed could have been spent on benchmarking additional SDN controllers' load balancing algorithms.

5.1.6.1 Was the problem statement answered?

1. *How to setup a load balancing algorithm in SDN in order to achieve performance comparable to a physical load balancer.*

    (a) Setting up load balancing for SDN was covered in approach sections 3.4.1 and 3.4.2, they were then tested in Chapter 4, as there was no way of knowing beforehand how the different controllers performed. Different approaches were tested but clear limitations were discovered in the analysis in Section 4.5. In this section, software limitations were discovered that did not allow for a setup that could compare with the performance and switching capabilities of the BIG-IP 1600 series LTM. The results also show that the controllers' real limitation is their system CPU. So if the controller resources can scale up to an LTM device in processing power, solving the software limitations would lead to a comparable SDN-based load balancing solution. Unfortunately, writing a better reactive algorithm than what the developers of POX and Floodlight have produced seemed unachievable in the allocated time-frame.

2. *Study, setup and run a physical load balancer (BIG-IP-1600 series).*

    (a) The provided load balancer from Jottacloud came pre-configured with unknown system passwords. Studying of service manuals and reconfiguring the device to interact with the Linux servers revealed that the device would only interact with VLAN tagged packets in the approach Section 3.2.1. As regard to the approach in retrospect, evaluating more controllers performance measured against each other instead of using a lot of time on configuring the physical environment would have been more beneficial for the SDN research field. This is because the results gathered from the physical environment don't have a physical counterpart SDN-based device to be compared too. The results however and the process of obtaining them are useful as the methods could be applied to the understanding and benchmarking of the virtual systems.

3. *Compare an SDN load balancer to a hardware load balancer in terms of scalability and performance.*

   (a) Performance:

      i. In Section 4.3 a performance graph for the physical environment was produced as a result of multiple performance tests. This shows a linear curve for using load balancing and that the CPU is the limiting factor. The same tests where therefore conducted on the virtual environment with the properties of the physical environment as a reference point. Analyses of these results in Section 4.5n shows that in some cases the SDN controllers may also achieve a linear performance curve similar to the physical unit.

   (b) Scalability:

      i. In terms of scalability, the 4.5 section shows that scaling of SDN load balancing is dependent on the controller. This means that the SDN-based solutions would scale as long as the controller could get more CPU power. This however seems more likely for the POX controller, as the modules are independent. Meaning that it is possible to turn of forwarding in POX, but if forwarding is turned off in Floodlight, the load balancing module does not work.

5.2 Recommendations for Future Work

This thesis ended up with good data for the physical environment and for two SDN-controllers in an automated Mininet environment. This also resulted in some predictions about the feasibility of an SDN-based load balancing, but as the SDN topic is wast and fast growing, covering all cases were not possible. However, based on the collated information some insight into good opportunities for further work in terms of another thesis, research papers or small projects is presented below:

As of now the tested controllers POX and Floodlight does not provide the same ease of operation as the BIG-IP delivers. Although the tests show that they can deliver similar performance in some scenarios. We cannot conclusively say that SDN can replace HTTP load balancing for small requests. Using a proactive setup the approach is possible and promising but it is probably not a wanted scenario for most users. For future work testing commercial controllers in the same scenarios could yield results, but are most likely prone to the same limits as the tested controllers. For conclusive results however, rerunning the tests on a physical OpenFlow enabled switch in a full-scale environment should be conducted in order to determine if the controller CPU load actually does impact the performance

as heavily as in the virtual environment.

From the results gathered, it is clear that processing power is the main limitation of an SDN-based load balancing. This means that expanding the work into more powerful controller applications would be a good research field. An example of this is an SDN controller that can run in a cluster for more processing power. This could be made possible with the Helium OpenDaylight SDN cluster and a OpenFlow switch that supports the OpenFlow 1.3.2 specification *OFPCR_ROLE_EQUAL*, which can set multiple controllers to control one switch. Another possibility is dividing the OpenFlow switch into different slices for different network traffic, like one slice for TCP (HTTP) and the other for IP switching with the FlowVisor software. This may possibly open up opportunities to either divide the roles of the controllers so that only one controller handles load balancing or increase the total throughput for the controller's load balancing algorithm. Either solution is an interesting topic for future research, as they both may make SDN load balancing viable.

There are many SDN controllers to chose from that supports load balancing, and while some are commercial grade like Vyatta, not all are. For instance is the OpenDaylight controller not commercial, and it comes bundled with a load balancing service. This means that it is possible to test even more controllers to help determining if some controllers can replace a dedicated unit. However, does Mininet not provide a direct comparison between a physical unit as it a virtualization. So to fully compare the SDN controllers to a physical unit; using an OpenFlow enables switch would enable that comparison. Unfortunately it was not an option during this thesis but it could be for other future projects. This means that both approaches are good future projects and if exploring additional controllers in Mininet, the high-level CLI script that has been developed in this thesis, is a good tool for benchmarking as only small changes would be required to support other controllers.

Bibliography

Armor, C. (n.d.). F5-big-ltm-1600. Retrieved February 3, 2015, from http://www.corporatearmor.com/

Asai, H., Fukuda, K., & Esaki, H. (2011). Traffic causality graphs: profiling network applications through temporal and spatial causality of flows. In *Proceedings of the 23rd international teletraffic congress* (pp. 95–102). ITC '11. San Francisco, California: International Teletraffic Congress. Retrieved from http://dl.acm.org/citation.cfm?id=2043468.2043484

Azodolmolky, S. (2013). *Software defined networking with openflow*. PACKT.

Chua, R. (2012). Nox pox and controllers galore murphy mccauley interview. Retrieved January 29, 2015, from https://www.sdxcentral.com/articles/interview/nox-pox-controllers-murphy-mccauley/2012/09/

CISCO. (2015). What is a network switch vs. a router? Retrieved January 26, 2015, from http://www.cisco.com/cisco/web/solutions/small_business/resource_center/articles/connect_employees_and_offices/what_is_a_network_switch/index.html

Cole, A. (2013). The end of the road for the vlan? not quite. Retrieved February 2, 2015, from http://www.enterprisenetworkingplanet.com/datacenter/datacenter-blog/end-of-the-road-for-the-vlan.html

Contemporary Control Systems, I. (2002). Hubs versus switches—understand the tradeoffs. *theEXTENSION*, *3*. Retrieved February 10, 2015, from http://www.ccontrols.com/pdf/Extv3n3.pdf

Edelman, J. (2013). Are there alternatives to the openflow protocol? Retrieved January 30, 2015, from http://searchsdn.techtarget.com/answer/Are-there-alternatives-to-the-OpenFlow-protocol

F5. (n.d.). Platform guide: 1600. Retrieved February 3, 2015, from https://support.f5.com/content/kb/en-us/products/big-ip_ltm/manuals/product/pg_1600/_jcr_content/pdfAttach/download/file.res/Platform_Guide__1600.pdf

F5. (2015). Load balancer. Retrieved February 3, 2015, from https://f5.com/glossary/load-balancer

Feamster, N., Rexford, J., & Zegura, E. (2013, December). The road to sdn. *Queue*, *11*(12), 20:20–20:40. doi:10.1145/2559899.2560327

Floodlight, P. (2015). Floodlight is an open sdn controller. Retrieved February 21, 2015, from http://www.projectfloodlight.org/floodlight/

Ghaffarinejad, A. & Syrotiuk, V. (2014, March). Load balancing in a campus network using software defined networking. In *Research and educational experiment workshop (gree), 2014 third geni* (pp. 75–76). doi:10.1109/GREE.2014.9

Handigol, N., Seetharaman, S., Flajslik, M., Johari, R., & McKeown, N. (2010, November). Aster*x: load-balancing as a network primitive. 9th GENI Engineering Conference (Plenary).

Handigol, N., Seetharaman, S., Flajslik, M., McKeown, N., & Johari, R. (2009). Plug-n-serve: load-balancing web traffic using openflow. Sigcomm Demonstration.

Iperf. (2014). What is iperf? Retrieved January 28, 2015, from https://iperf.fr/

Jesse. (2015). Bip-ip 1600 vlan bridge performance degradation. Retrieved March 8, 2015, from https://devcentral.f5.com/questions/bip-ip-1600-vlan-bridge-performance-degradation

Johnson, S. (2012). A primer on northbound apis: their role in a software-defined network. Retrieved January 30, 2015, from http://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network

Laboratories, H.-P. R. (n.d.-a). 2 an example of using httperf. Retrieved from http://www.hpl.hp.com/research/linux/httperf/wisp98/html/doc003.html

Laboratories, H.-P. R. (n.d.-b). Httperf(1) - linux man page. Retrieved from http://linux.die.net/man/1/httperf

Limoncelli, T. A. (2012, June). Openflow: a radical new idea in networking. *Queue*, *10*(6), 40:40–40:46. doi:10.1145/2246036.2305856

McCauley, J. (2013). Arp responder.py. Retrieved May 4, 2015, from https://github.com/CPqD/RouteFlow/blob/master/pox/pox/misc/arp_responder.py

McCauley, M. & Al-Shabibi, A. (2014). Pox wiki. Retrieved February 7, 2015, from https://openflow.stanford.edu/display/ONL/POX+Wiki

McKeown, N. (2011). How sdn will shape networking. Retrieved February 9, 2015, from https://www.youtube.com/watch?v=c9-K5O_qYgA

Morin, J. (2014). Conflicting trends part 2: sdn and converged infrastructure. Retrieved January 26, 2015, from http://www.extremenetworks.com/conflicting-trends-part-2-sdn-and-converged-infrastructure

NOXRepo.org. (2015). About pox. Retrieved January 29, 2015, from http://www.noxrepo.org/pox/about-pox/

Oliver, B. (2014). Pica8: first to adopt openflow 1.4; why isn't anyone else? Retrieved February 7, 2015, from http://www.tomsitpro.com/articles/pica8-openflow-1.4-sdn-switches,1-1927.html

open networking foundation. (2015). Software-defined networking (sdn) definition. Retrieved January 30, 2015, from https://www.opennetworking.org/sdn-resources/sdn-definition

OpenFlow. (2009). Openflow switch specification. Retrieved February 4, 2015, from http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf

OpenFlow. (2013). Openflow switch specification 1.3.3. Retrieved February 4, 2015, from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf

OpenFlow. (2014). Openflow switch specification 1.5. Retrieved February 4, 2015, from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf

Pacchiano, R. (2006). The difference between hubs, switches and routers. Retrieved January 27, 2015, from http://www.webopedia.com/DidYouKnow/Hardware_Software/router_switch_hub.asp

Phaal, P. (2013). Sdn and large flows. Retrieved February 3, 2015, from http://blog.sflow.com/2013/02/sdn-and-large-flows.html

Pronschinske, M. (2013). Controllers q&a: nox/pox , trema , beacon , floodlight , routeflow. Retrieved January 26, 2015, from http://architects.dzone.com/articles/controllers-qa-noxpox-trema

Richard Wang, J. R., Dana Butnariu. (2011). Openflow-based server load balancing gone wild. *Princeton University*.

Rouse, M. (2007). Hardware load-balancing device (hld). Retrieved February 3, 2015, from http://searchnetworking.techtarget.com/definition/hardware-load-balancing-device

Salisbury, B. (2012). Tcams and openflow – what every sdn practitioner must know. Retrieved February 7, 2015, from https://www.sdxcentral.com/articles/contributed/sdn-openflow-tcam-need-to-know/2012/07/

Team, M. (2014). Introduction to mininet. Retrieved February 23, 2015, from https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

Team, M. (2015). Mininet overview. Retrieved January 28, 2015, from http://mininet.org/overview/

Uppal, H. & Brandon, D. (n.d.). Openflow based load balancing. University of Washington - Project Report CSE561: Networking. Retrieved February 11, 2015, from http://courses.cs.washington.edu/courses/cse561/10sp/project_files/cse561_openflow_project_report.pdf

Wang, K.-C. (2012). Load balancer. Retrieved February 22, 2015, from http://www.openflowhub.org/display/floodlightcontroller/Load+Balancer

Wireshark. (2015). About. Retrieved January 28, 2015, from https://www.wireshark.org/about.html

Zito, P. (2013). What is the osi model. Retrieved January 27, 2015, from http://blog.buildingautomationmonthly.com/what-is-the-osi-model/

Appendices

Appendix A

Mininet scripts

Listing A.1: POX Mininet

```
#!/usr/bin/python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.node import Controller

from mininet.link import TCIntf
from mininet.util import custom, quietRun
from mininet.node import CPULimitedHost
from mininet.log import setLogLevel, info



from functools import partial
import os

class SingleSwitchTopo(Topo):
        "Single switch connected to n hosts."
        def build(self, n=2):
                switch = self.addSwitch('s1')
                for h in range(n):
                        host = self.addHost('h%s' % (h + 1))
                        # Define network parameters like delay and
                            ↪ speed:
                        self.addLink(host, switch, bw=820, delay='0.07
                            ↪ ms', loss=0, use_htb=True)


class POXload ( Controller ):
        "Controller class to start the controller"
        def start( self ):
                "Start POX component of your choise:"
                self.pox = '%s/pox/pox.py' % os.environ[ 'HOME' ]
                # Forwarding:
                #self.cmd( self.pox, 'forwarding.l2_learning &' )
                # Reactive Load balance:
                #self.cmd( self.pox, 'misc.ip_loadbalancer --ip
                    ↪ =10.1.2.3 --servers=10.0.0.2,10.0.0.3,10.0.0.4
                    ↪ &' )
                # Proactive Load balance:
                self.cmd( self.pox, 'load arp_responder
                    ↪ --10.0.0.10=00:00:00:00:10:fe forwarding.
                    ↪ l2_learning &' )
        def stop( self ):
                "Stop POX"
                self.cmd( 'kill %' + self.pox )

controllers = { 'poxload': POXload }

def startstuff():
        # Define topology (I.E number of hosts):
        topo = SingleSwitchTopo(n=4)
        # Define CPU limits for hosts:
        host = custom( CPULimitedHost, sched='cfs', period_us=10000,
            ↪ cpu=.025 )
        # Define network:
        #  autoSetMacs=True is equivalent to --mac parameter
        net = Mininet(topo=topo, link=TCLink, controller=POXload( 'c0
            ↪ ', ip='127.0.0.1', port=6633 ), host=host, autoSetMacs=
            ↪ True )
        # Start network:
        net.start()
        # Start terminals
        #net.startTerms()
```

```
58
59          # Start iperf servers on h3 & h4 (servers)
60          h3 = net.get('h3')
61          #h3.cmd('iperf -s &')
62          #h3.cmd('http-server -p 80 -s &')
63
64          h4 = net.get('h4')
65          #h4.cmd('iperf -s &')
66          #h4.cmd('http-server -p 80 -s &')
67
68          # Insert benchmark command here:
69          # For instance like so:
70          h1 = net.get('h1')
71          # h1.cmd('httperf 10.0.0.10 > logfile-httperf.txt')
72          # h1.cmd('ab 10.0.0.10 > logfile-ab.txt')
73          # Can be expanded into a more automated script looking for
                ↪ saturation,
74          # for instance to run a separate saturation finding script:
75          # h1.cmd('./saturation.py -s 10.0.0.10 -u "/index.html" -c 1 -
                ↪ o POX-forwardTest.dat')
76
77          # Starts the CLI, remove to quit the network after this line.
                ↪ (for example if automated test is done):
78          CLI(net)
79          # Kills the network
80          net.stop()
81
82  if __name__ == '__main__':
83          setLogLevel('info')
84          startstuff()
```

Listing A.2: Floodlight Mininet

```
#!/usr/bin/python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.node import Controller

from mininet.link import TCIntf
from mininet.util import custom, quietRun
from mininet.node import CPULimitedHost
from mininet.log import setLogLevel, info


from functools import partial
import os

class SingleSwitchTopo(Topo):
        "Single switch connected to n hosts."
        def build(self, n=2):
                switch = self.addSwitch('s1')
                for h in range(n):
                        host = self.addHost('h%s' % (h + 1))
                        # Define network parameters like delay and speed:
                        self.addLink(host, switch, bw=820, delay='0.07 ms', loss=0, use_htb=True)

class Floodload ( Controller ):
        def start( self ):
                "Start Floodlight controller"
                self.pox = 'bash %s/floodlight/floodlight.sh' % os.environ[ 'HOME' ]
                self.cmd( self.pox, '' )
        def stop( self ):
                "Stop POX"
                self.cmd( 'kill %' + self.pox )

controllers = { 'floodload': Floodload }

def startstuff():
        # Define topology (I.E number of hosts):
        topo = SingleSwitchTopo(n=4)
        # Define CPU limits for hosts:
        host = custom( CPULimitedHost, sched='cfs', period_us=10000, cpu=.025 )
        # Define network:
        net = Mininet(topo=topo, link=TCLink, controller=Floodload( 'c0', ip='127.0.0.1', port=6653 ), host=host )
        # Start network:
        net.start()
        # Start terminals:
        #net.startTerms()

        # Start iperf servers on h3 & h4 (servers)
        h3 = net.get('h3')
        #h3.cmd('iperf -s &')
        #h3.cmd('http-server -p 80 -s &')

        h4 = net.get('h4')
        #h4.cmd('iperf -s &')
        #h4.cmd('http-server -p 80 -s &')

        # Insert benchmark command here:
        # For instance like so:
        h1 = net.get('h1')
```

```
62          # h1.cmd('httperf 10.0.0.10 > logfile-httperf.txt')
63          # h1.cmd('ab 10.0.0.10 > logfile-ab.txt')
64          # Can be expanded into a more automated script looking for
                ↪ saturation.
65
66          # Starts the CLI, remove to quit the network after this line.
                ↪ (for example if automated test is done):
67          CLI(net)
68          # Kills the network
69          net.stop()
70
71
72          CLI(net)
73          net.stop()
74
75  if __name__ == '__main__':
76          setLogLevel('info')
77          startstuff()
```

Listing A.3: Load Component POX

```
#!/usr/bin/python
'''
Parameter for arp_responder is to bind a Virtual IP to a HW addr.
Start with ./pox.py load arp_responder --10.0.0.10=00:00:00:00:10:fe
    ↪ forwarding.l2_learning  log.level --DEBUG

'''

# Import some POX stuff
from pox.core import core                     # Main POX object
import pox.openflow.libopenflow_01 as of      # OpenFlow 1.0 library
from pox.lib.addresses import EthAddr, IPAddr # Address types
from pox.lib.util import dpid_to_str
from pox.lib.revent import *


log = core.getLogger()

# When connection is established to the switch:
class ConnectionUp(Event):
    def __init__(self,connection,ofp):
                Event.__init__(self)
                self.connection = connection
                self.dpid = connection.dpid
                self.ofp = ofp

                # Pushes static rule to controller:
                # For client->server 1
                msg = of.ofp_flow_mod()

                # Subnet input: Can be any IP with it's Prefix
                # I.E: 10.0.0.0/8, 10.0.0.0/24, 192.168.0.1,
                msg.match.nw_src = "10.0.0.0/8"

                msg.priority = 65535            # Rule Priority
                msg.match.dl_type = 0x800       # IPv4

                msg.match.nw_dst = IPAddr("10.0.0.10")  # Virtual IP

                msg.match.nw_proto = 6 # TCP
                msg.match.tp_dst = 80  # HTTP

                # Rewrite MAC addr, IP addr and output on port server
                    ↪ is connected to:
                msg.actions.append(of.ofp_action_dl_addr.set_dst(
                    ↪ EthAddr("00:00:00:00:00:03")))
                msg.actions.append(of.ofp_action_nw_addr.set_dst(
                    ↪ IPAddr("10.0.0.3")))
                msg.actions.append(of.ofp_action_output(port = 3)) #
                    ↪ Must be last parmeter

                # Send rule to server:
                self.connection.send(msg)


                # For server->client 1. More a general rule as it will
                    ↪  rewrite all TCP80
                # from the server serving the virtual IP.
                self.connection = connection
                self.ofp = ofp
                msg = of.ofp_flow_mod()

                msg.match.nw_src = "10.0.0.3"
                msg.priority = 65535
                msg.match.dl_type = 0x800
                msg.match.dl_src = EthAddr("00:00:00:00:00:03")
```

```
61
62                  msg.match.nw_proto = 6
63                  msg.match.tp_src = 80
64
65                  msg.match.in_port = 3
66
67                  msg.actions.append(of.ofp_action_dl_addr.set_src(
                        ↪ EthAddr("00:00:00:00:00:10")))
68                  msg.actions.append(of.ofp_action_nw_addr.set_src(
                        ↪ IPAddr("10.0.0.10")))
69                  msg.actions.append(of.ofp_action_output(port = 1))
70
71                  self.connection.send(msg)
72
73  class ConnectionDown(Event):
74          def __init__(self,connection,ofp):
75                  Event.__init__(self)
76                  self.connection = connection
77                  self.dpid = connection.dpid
78
79  class MyComponent(object):
80          def __init__(self):
81                  core.openflow.addListeners(self)
82
83          def _handle_ConnectionUp(self,event):
84                  ConnectionUp(event.connection,event.ofp)
85                  log.info("Switch␣%s␣has␣come␣up,␣and␣installed␣the␣
                        ↪ rules",dpid_to_str(event.dpid))
86
87          def _handle_ConnectionDown(self,event):
88                  ConnectionDown(event.connection,event.dpid)
89                  log.info("Switch␣%s␣has␣shutdown.",dpid_to_str(event.
                        ↪ dpid))
90
91  def launch():
92          core.registerNew(MyComponent)
```

Appendix B

Raw data

For some tests where the data was collected out of normal 3 test average scenario, their result is presented here:

Table B.1: Proactive POX httperf test of 1 server-raw

	Httperf: (60*R=Num-conns)	Request rate asked: (req/s)	Compleded rate	Duration
1	2520	42	41,1	61,3
2	2520	42	40,9	61,6
3	2520	42	41	61,4
4	2520	42	42,2	61,2
5	2520	42	40,8	61,7
6	2520	42	41,4	60,9
7	2520	42	41,1	61,3
8	2520	42	41,8	60,2
9	2520	42	41,1	61,3
Standard deviation			0,355555556	0,296296296
90% CI interval (t.test)			40,98262-41,5507	60,93661-61,48561
Mean			41,26667	61,21111

Table B.2: Proactive POX httperf test of 2 servers-raw

	Httperf: (60*R=Num-conns)	Request rate asked: (req/s)	Compleded rate	Duration
1	2520	42x2	83,5	60,31
2	2520	42x2	83,6	60,38
3	2520	42x2	83,5	60,29
4	2520	42x2	82	60,24
5	2520	42x2	83,8	60,37
6	2520	42x2	83,6	60,22
7	2520	42x2	83,6	60,16
8	2520	42x2	83,4	60,13
1				60,1
2				60,2
3				60,34
4				60,35
5				60,24
6				60,26
7				60,45
8				60,36
Standard deviation			0,34375	0,080622222
90% CI interval (t.test)			82,99482-83,75518	60,23185-60,31815
Mean			83,375	60,275